RECHERCHE ET ANALYSE DE SOLUTIONS PERFORMANTES POUR LE TRAITEMENT DE FICHIERS JSON DANS UN LANGAGE DE HAUT NIVEAU

MÉMOIRE PRÉSENTÉ COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN TECHNOLOGIE DE L'INFORMATION

PAR
CARINE CROTEAU

JANVIER 2021

REMERCIEMENTS

Je tiens d'abord à remercier mon professeur encadrant, M. Daniel Lemire, qui a su me guider tout au long de mon cheminement de maîtrise. M. Lemire a su trouver les stratégies me permettant de me dépasser, et m'a offert de judicieux conseils dans mon projet de recherche. Les orientations qu'il m'a proposées me permettent de présenter des résultats qui vont au-delà de ce que j'aurais pu espérer au départ. Son esprit critique et son envie de toujours aller plus loin ont été une source de motivation constante dans mon cheminement. Je suis certaine que cette motivation continuera de me contaminer dans de futurs projets de recherche.

Je remercie également l'université TÉLUQ pour leur soutien administratif ainsi que pour les accès aux équipements qui me permettent de présenter dans ce mémoire des résultats justes, au summum des performances matérielles possibles. Sans un tel soutien, il m'aurait été impossible de terminer mes études dans les délais que je m'étais fixés. Je remercie également l'université pour la qualité des enseignements et des suivis à distance que j'ai reçus tout au long de mon parcours de maîtrise.

Je tiens aussi à remercier la Fédération des enseignantes et enseignants de cégep (FEC-CSQ) de m'avoir permis de réduire ma charge de travail comme enseignante pour compléter ma maîtrise. Sachez que je suis tout à fait consciente de l'opportunité qui m'a été offerte et que j'envisage assurément de rapporter plusieurs de ces acquis dans la formation des futures générations de demain. J'en profite pour féliciter tous les étudiantes et étudiants ayant terminé des études de deuxième ou troisième cycle. J'encourage aussi toutes les personnes qui travaillent fort en ce moment pour poursuivre des études de haut niveau, je sais que vous pouvez y arriver.

Finalement, j'aimerais prendre un moment pour remercier mes deux enfants, Alexis et Coralie, qui m'ont trop de fois entendu "Attend un peu, maman doit travailler" afin de me permettre de respecter les délais que je m'étais fixés. Je sais, à 4 et 6 ans, ce n'est pas toujours facile de comprendre. Sachez que je suis fière de vous, en cette situation particulière de 2020 où nous avons dû rester à la maison pour notre santé ainsi que celle des autres et nous adapter constamment, vous êtes des champions. Merci de m'avoir permis d'avancer, parfois à pas de tortue, dans cette grande aventure qu'est la rédaction d'un mémoire. Malgré les circonstances, ne vous inquiétez pas, ça va bien aller.

RÉSUMÉ

Nous sommes en 2020, cette époque où nous collectons plus d'information que nous ne pouvons en traiter. De nombreuses recherches visent à augmenter la rapidité du traitement de l'information, tant au niveau des architectures matérielles que logicielles. Nous sommes aussi à une époque où l'échange d'information entre divers systèmes est omniprésent. De multiples appareils sont exploités par les utilisateurs, ce qui rend la tâche des programmeurs de plus en plus complexe.

Parmi cette réalité se dégage une forte tendance : l'échange d'information au format de données textuelles JavaScript Object Notation (JSON), un dérivé de la notation des objets du langage JavaScript. Ce format de données, facile à comprendre par l'utilisateur, propose une syntaxe permettant à l'ordinateur d'en faire l'analyse à des fins d'affichage, de calculs statistiques, d'analyses approfondies ou pour répondre à tout autre besoin. Dans une ère de données massives et d'intégration de l'intelligence artificielle à tous les niveaux, les possibilités sont infinies.

Pour traiter de plus en plus d'information, il importe d'améliorer les algorithmes d'analyse syntaxique. Le présent mémoire s'appuie sur des recherches sur l'analyseur syntaxique de la bibliothèque `simdjson` développé en C++ qui présente en ce moment un vif intérêt par la communauté informatique. L'algorithme proposé par la bibliothèque `simdjson` repose sur l'exploitation des capacités de parallélisme des nouvelles architectures matérielles dans le but d'optimiser les performances de l'analyse syntaxique de documents au format JSON. Le défi présenté dans ce mémoire repose sur l'intégration de ses fonctionnalités dans le langage de haut niveau JavaScript dans le but d'élargir l'accès à ces fonctions d'analyse à une plus grande proportion de développeurs.

Afin de résoudre la problématique, deux stratégies, appuyées sur des travaux existants, ont été évaluées. La première repose sur le principe du chargement paresseux, alors que la seconde repose sur le principe d'encapsulation d'un objet dans un module intermédiaire. Les expérimentations ont permis d'obtenir, selon diverses méthodes, des vitesses de 3 à 7 fois plus rapides que la fonction d'analyse syntaxique nativement intégrée dans la plateforme sélectionnée. Nous espérons que ce mémoire contribuera au développement de futurs travaux en lien avec l'optimisation des performances des analyseurs syntaxiques actuellement utilisés.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES EXTRAITS DE CODE

INTRODUCTION

Le traitement et l'échange de données massives dans un temps optimal sont un enjeu crucial pour le développement actuel et futur de logiciels de classification, de recherche et d'intelligence artificielle. La notation d'objet JavaScript (JavaScript Object Notation - JSON) étant actuellement la plus utilisée pour ce type d'échanges, il importe de trouver des solutions pour en maximiser le traitement.

Afin de pouvoir échanger des données entre plusieurs applications, le format JSON est souvent adopté et fait désormais partie de la norme puisqu'il présente des données semi-structurées dans un format flexible, facile à lire et à comprendre. Cependant, ces avantages ont un coût : le temps de conversion du contenu en langage machine. Pour répondre à ce besoin, la majorité des langages de programmation implémentent une fonction d'analyse syntaxique. Cependant, avec les volumes de données que les ordinateurs doivent actuellement traiter, surtout dans un contexte de stockage de données massives et de l'utilisation de ces données à des fins d'analyse ou d'intelligence artificielle, les performances des diverses méthodes d'analyse syntaxique représentent une réelle problématique.

En juin 2017, Microsoft Research a publié un article sur un analyseur syntaxique JSON nommé Mison (Li *et al.*, 2017). Cet article affirme qu'il est possible d'obtenir un gain de performances considérable des analyseurs syntaxiques les plus couramment utilisés. Dans l'article en question, Li et al. expliquent tirer avantage des nouveaux processeurs et de leur architecture SIMD (Single instruction on Multiple Data), qui permettent d'exécuter une même instruction sur de multiples données en utilisant les capacités de parallélisme du processeur. L'article présente

des résultats intéressants ainsi qu'un sommaire des structures de données et de l'algorithme employés, mais le code source n'a pas été publié.

En février 2019, Langdale et Lemire ont repris le concept présenté par l'article et développé un algorithme en C++ qui a obtenu des performances supérieures aux bibliothèques d'analyses syntaxiques existantes. L'article "Parsing Gigabytes of JSON per Second" (Langdale et Lemire, 2019) présente le résultat de leurs démarches.

Bien que les résultats soient prometteurs, un autre défi se présente désormais. Selon un sondage annuel (Stack Overflow, 2020) complété par 65 000 développeurs logiciels en provenance de 186 pays, moins du quart d'entre eux utilisent des langages de programmation de bas niveau comme C (21.8%), C++ (23.9%). Les langages les plus couramment utilisés sont JavaScript (67.7%), Python (44.1%), Java (40.2%), C# (31.4%) et PHP (26.2%).

CHAPITRE I

PROBLÉMATIQUE

Le présent mémoire s'inscrit dans un contexte où le développement Web est omniprésent et que l'utilisation d'interfaces de programmation applicatives utilisant la notation d'objet JavaScript (JavaScript Object Notation - JSON) est de plus en plus répandue, principalement dans le développement d'applications Web manipulant des données massives. Les expérimentations présentées s'inscrivent dans la poursuite des travaux en cours sur la bibliothèque `simdjson` proposant un analyseur syntaxique haute performance développé en C++, et présentant actuellement un vif intérêt. Elles visent à valider si l'intégration de cette bibliothèque dans le langage de haut niveau JavaScript permet d'offrir de meilleures performances que la méthode d'analyse syntaxique native `JSON.parse()`, ainsi qu'à évaluer la perte de performance que cela engendre en comparaison à l'utilisation native de la bibliothèque `simdjson`.

S'appuyant sur des recherches en cours visant à offrir des techniques d'analyse syntaxique toujours plus performantes ainsi que sur les meilleures pratiques en termes de développement d'extensions durables, il est possible que d'autres solutions plus performantes soient offertes dans un avenir rapproché. Nous espérons que les travaux présentés contribueront à l'amélioration continuelle des techniques de traitement de l'information.

1.1 Définitions et objet de recherche

Afin de bien cerner la problématique présentée, il est important de bien comprendre les méthodologies de développement Web actuelles et les principes fondamentaux portant sur l'échange de données, le fonctionnement d'une architecture SIMD et son application dans la bibliothèque `simdjson`. Les prochaines sections proposent une description des éléments fondamentaux considérés dans ce projet de recherche.

1.1.1 Développement Web

Depuis la première annonce publique du World Wide Web en 1990, le développement d'applications a grandement évolué. Les données sont désormais majoritairement stockées sur des serveurs distants afin de les rendre accessibles de partout. Le nombre d'applications Web est en constante augmentation, ce qui fait en sorte qu'une proportion significative de développeurs utilisent des technologies Web pour répondre aux besoins à combler. Avec l'arrivée des appareils mobiles, le développement natif a effectué un regain, obligeant au départ le développement de plusieurs versions d'une même application pour permettre son utilisation sur plusieurs types d'appareils. Avec le temps, des outils de déploiement ont été produits pour permettre le développement d'une seule application pouvant être déployée sur plusieurs types d'appareils.

1.1.1.1 Applications Web progressives

Dans les dernières années, une nouvelle tendance fait surface : le développement d'applications Web mobiles. Ces applications ressemblent à une application native, mais sont exécutées dans le navigateur de l'appareil, au même titre qu'une application Web utilisée sur un ordinateur. Plus communément appelées applications

Web progressives (Progressive Web Apps ou PWA), elles sont de plus en plus supportées par les plus grands leaders technologiques. C'est notamment le cas de Microsoft qui a intégré l'utilisation de celles-ci dans son navigateur officiel Edge ainsi que dans le système d'exploitation Windows 10 (Pflug *et al.*, 2018). Google permet également depuis le début de l'année 2019 la distribution d'applications progressives dans son magasin d'applications Google Play (Mclachlan, 2019). Plusieurs applications mobiles fréquemment utilisées sont désormais disponibles sous ce format. Parmis celles-ci se trouvent certains outils proposés par Google (Google Drive, Photos, Youtube Music, Stadia, etc.), ainsi que d'autres applications populaires telles que Twitter (Gallagher, 2017), Outlook (Warren, 2019), Spotify et Instagram.

1.1.1.2 Architecture microservices

Afin d'offrir plusieurs variantes de leurs applications en fonction du type d'appareil et de la taille de l'écran utilisés par l'utilisateur, et de permettre la réutilisation de composants entre diverses applications d'une même suite ou d'un même fournisseur, plusieurs équipes de développement ont adopté la technique de développement de microservices. Cette technique, représentée à la figure 1.1, structure une application comme un ensemble de services faiblement couplés. Les microservices indépendants communiquent les uns avec les autres en utilisant des API indépendantes du langage de programmation, souvent à l'aide d'API REST.

Un avantage avancé par cette stratégie de développement est que ces microservices peuvent séparément être mis à jour tandis que l'application demeure compatible avec les autres services qu'elle exploite. La philosophie de l'architecture microservices s'inspire en grande partie de la philosophie UNIX : "Write programs that do one thing and do it well" (Raymond, 2003).

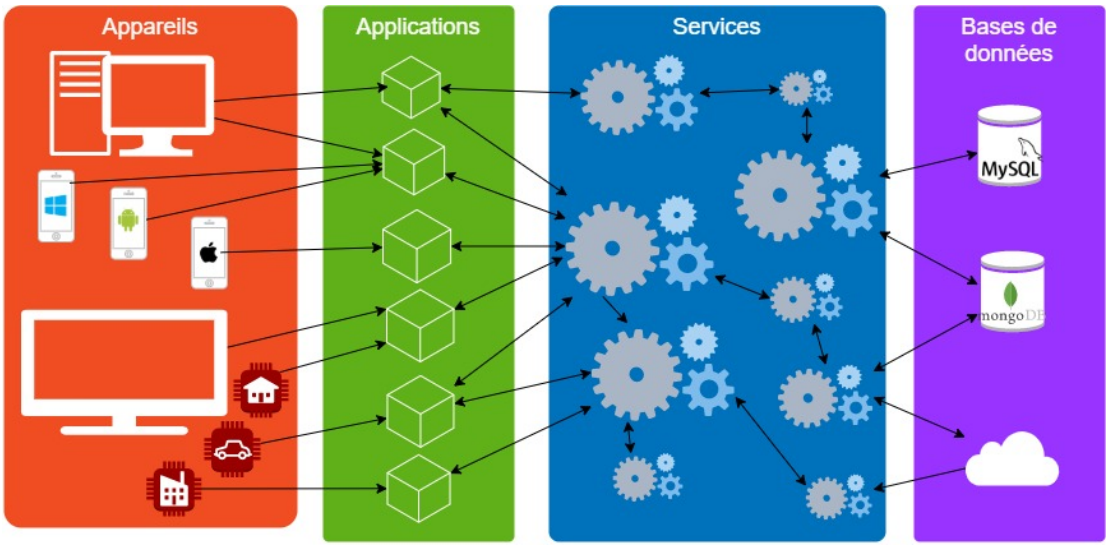

Figure 1.1: Technique de développement logiciel en microservices

1.1.1.3 Bibliothèques et cadriciels

Dans le cadre de développement d'applications Web, les développeurs sont souvent appelés à utiliser des algorithmes très similaires dans différents projets. Au fil des ans, plusieurs bibliothèques ont ainsi été développées et offertes à la communauté de développeurs afin d'offrir des fonctionnalités robustes, stables et uniformes aux utilisateurs du Web.

L'une des premières bibliothèque JavaScript massivement adoptée fût jQuery, lancée en 2006. Créée pour faciliter l'écriture de scripts côté client en réutilisant le principe des sélecteurs CSS, la bibliothèque jQuery permet ainsi de manipuler dynamiquement le contenu d'une page HTML, plus communément nommé DOM HTML (Document Object Model), plus facilement. En 2007 s'ajoute jQuery UI, une bibliothèque offrant des composants visuels, nommés widgets, ainsi que des thèmes pouvant être implémentés avec jQuery. Ces deux bibliothèques ont grandement contribué à l'évolution du Web tel qu'on le connaît aujourd'hui.

En 2009, Miško Hevery, employé de chez Google, développe un projet parallèle dans le but de faciliter le développement d'application Web pour certains projets internes sur lequel il travaille. Le projet prend de l'ampleur et est réutilisé par des collègues, toujours pour des projets internes. En 2010, le projet est publié en tant que projet open source. D'abord connu sous le nom d'AngularJS, il a depuis été repris par Google. En 2014, Google publie la version 2.0, nommée Angular, qui propose des changements significatifs qui ont fait l'objet de plusieurs critiques, le langage TypeScript remplaçant désormais JavaScript. Angular est considéré comme un cadriciel (framework) puisqu'il redéfinit complètement la structure d'une application Web en imposant le patron de conception Modèle-Vue-Controlleur (MVC).

En parallèle, en 2011, Jordan Walke, ingénieur informatique chez Facebook, crée React. La bibliothèque JavaScript permet d'accélérer le traitement du contenu HTML en intégrant un DOM virtuel plus facile à manipuler. Ainsi, plutôt que de générer le DOM directement en HTML, une arborescence représentant le DOM est d'abord chargée en mémoire, ce qui rend sa manipulation plus rapide. Pete Hunt, ingénieur travaillant sur Instagram, s'intéresse au projet et collabore avec Jordan Walke afin de rendre le code source indépendant de Facebook, ce qui permet de publier une première version en 2013 sous licence Apache 2.0.

Plus récemment, en 2013, Evan You, ancien employé de chez Google, utilise AngularJS pour plusieurs de ses projets. L'idée lui vient alors de reprendre les meilleurs éléments d'Angular et de produire une solution plus légère. C'est ainsi que naît le cadriciel Vue.js en 2014. Ne bénéficiant pas de la même notoriété que ses prédécesseurs, Vue.js a connu un démarrage moins rapide que ceux-ci. Cependant, depuis juillet 2018, ce cadriciel connaît une meilleure ascension qu'Angular au niveau des téléchargements.

React, Vue.js et Angular représentent actuellement les cadriciels et bibliothèques les plus utilisés par les développeurs Web. Les statistiques de téléchargement de ces outils peuvent être consultées à la figure 1.2 (npm trends, 2021).

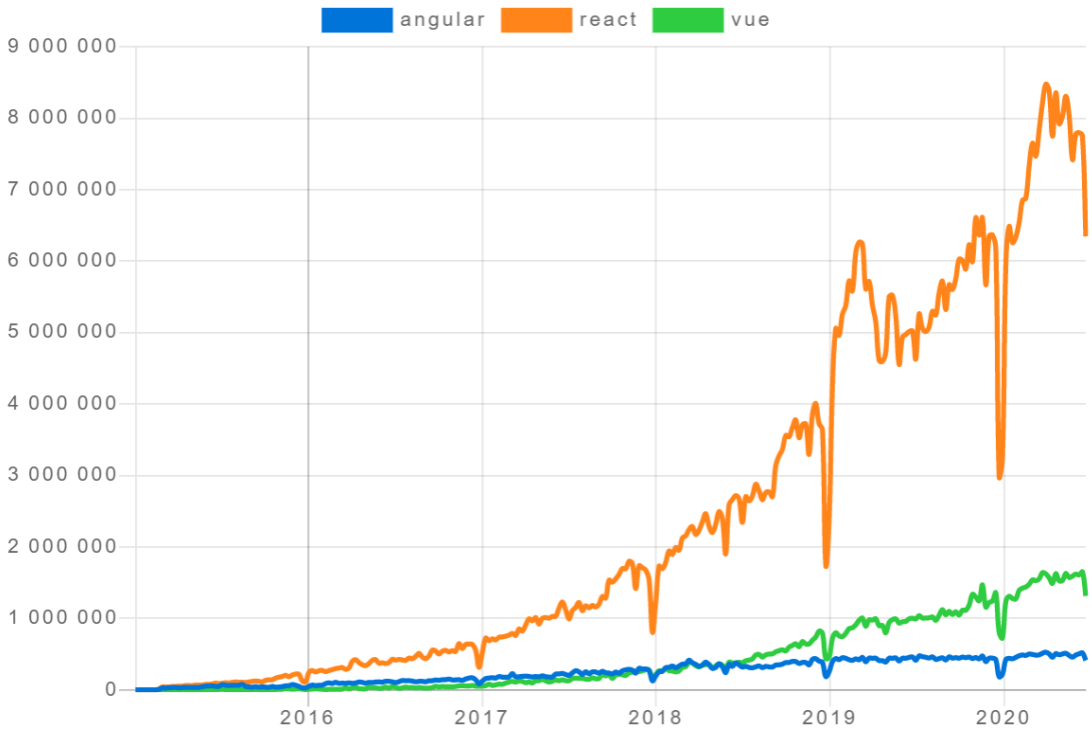

Figure 1.2: Téléchargements d'Angular, React et Vue.js depuis leur publication

1.1.2 Environnement JavaScript

JavaScript (qui est souvent abrégé en "JS") est un langage de script léger, orienté objet, principalement connu comme le langage de script des pages Web. Il est aussi utilisé dans de nombreux environnements extérieurs aux navigateurs Web tels que Node.js, Apache CouchDB et Adobe Acrobat. Le code JavaScript est interprété ou compilé à la volée (JIT). C'est un langage à objets utilisant le concept de prototype, disposant d'un typage faible et dynamique qui permet de programmer suivant

plusieurs paradigmes de programmation : fonctionnelle, impérative et orientée objet (MDN Web Docs, 2021a).

JavaScript est considéré comme un langage de programmation de haut niveau. Un langage de programmation de haut niveau a une abstraction significative des détails du fonctionnement de l'ordinateur. Il est conçu pour être facilement compris par les humains et pour cette raison, il doit être traduit par un autre logiciel. Contrairement aux langages de programmation de bas niveau, il peut utiliser des éléments de langage naturel ou automatiser (voire masquer) des champs importants du système informatique, rendant le processus de développement plus simple et plus compréhensible par rapport à un langage de niveau inférieur (MDN Web Docs, 2021b).

Cette abstraction, qui offre un net avantage au niveau de la facilité et de la rapidité à mettre en oeuvre une solution informatique, a cependant un coût. Les structures de données utilisées étant génériques et l'accès direct à la mémoire de l'ordinateur non possible, les programmes développés dans ce type de langage sont généralement moins performants et sont de grands consommateurs de mémoire. Pour exécuter du code JavaScript, un navigateur Web ou toute plateforme logicielle doit contenir un moteur JavaScript lui permettant d'interpréter puis d'exécuter du code. Le premier moteur JavaScript, SpiderMonkey, a été créé par l'informaticien américain Brendan Eich pour le navigateur Netscape Navigator (Rauschmayer, 2014). Depuis, chaque fournisseur de solutions exécutant du code écrit dans le langage JavaScript tente de concevoir le moteur qui sera le plus performant.

En 2008, Google publie ses premiers navigateurs Web, Google Chrome et Chromium, intégrant le moteur JavaScript V8 open source. V8 propose à l'époque une nouvelle méthode pour traiter un script JavaScript. Plutôt que d'utiliser des techniques plus traditionnelles telles que l'interprétation du code binaire ou la compilation du

programme complet en code machine et l'exécution à partir d'un système de fichiers, le moteur V8 compile directement le code JavaScript en code machine natif avant de l'exécuter. Le code compilé est ainsi optimisé (et réoptimisé) dynamiquement au moment de l'exécution, en fonction du profil d'exécution du code.

Les autres moteurs JavaScript les plus connus sont SpiderMonkey, reconnu pour son utilisation par le navigateur Web FireFox, JavaScript Core utilisé par le navigateur Web Safari et Chakra utilisé par le navigateur Web Microsoft Edge. Chaque moteur JavaScript utilise des techniques différentes d'optimisation de code afin d'offrir les meilleures performances possibles et de limiter le coût d'utilisation de ce langage de haut niveau. Il demeure difficile d'identifier le moteur le plus performant sans d'abord tenir compte de l'utilisation que fait le développeur du langage JavaScript.

À ce jour, la plateforme logicielle Node.js, permettant d'exécuter du code JavaScript sur un serveur, est bâtie sur le moteur JavaScript V8. Le système de gestion de bases de données MongoDB, permettant de manipuler des donnes structurées au format BSON (JSON binaire), utilisait initialement le moteur JavaScript V8. En décembre 2015, MongoDB est passé au moteur JavaScript SpiderMonkey, moteur étant toujours utilisé dans sa dernière version 4.2 publiée en août 2019.

1.1.2.1 Node.js

Node.js est une plateforme logicielle libre en JavaScript asynchrone piloté par les événements. Conçue pour créer des applications réseau évolutives. elle utilise la machine virtuelle V8, la librairie libuv pour sa boucle d'événements, et implémente sous licence MIT les spécifications CommonJS.

Concrètement, Node.js est un environnement bas niveau permettant l'exécution de JavaScript côté serveur. Node.js est utilisé notamment comme plateforme de serveur Web, et est reconnu pour faciliter le développement d'API ou d'applications utilisant

des données structurées au format JSON. La figure 1.3 présente un sommaire de l'architecture Node.js.

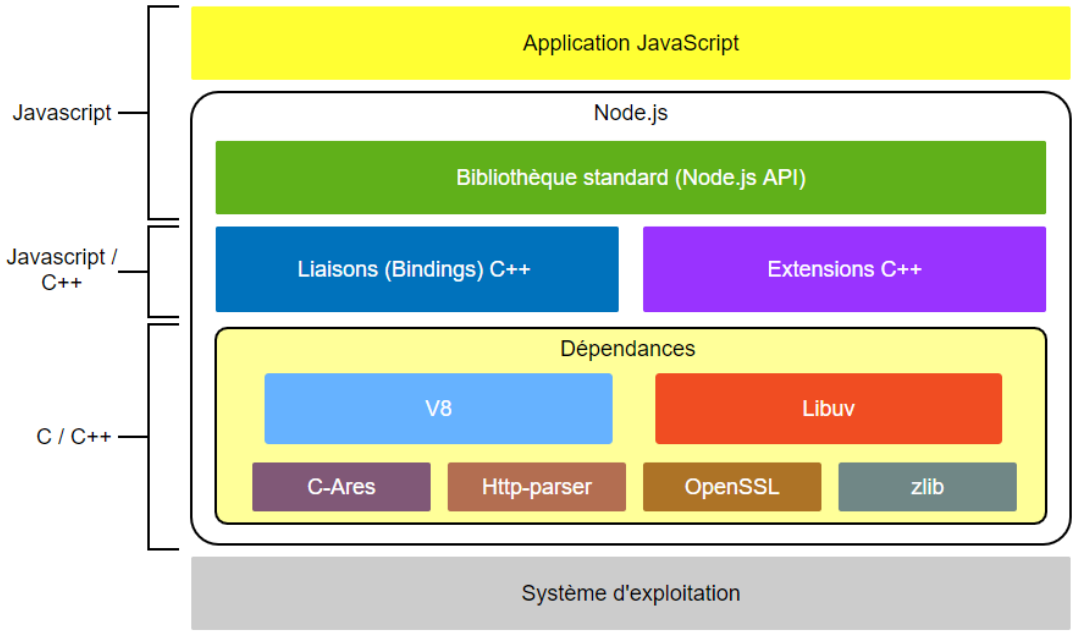

Figure 1.3: Architecture Node.js

Bien que la pile logicielle LAMP (Linux, Apache, MySQL et PHP) soit encore majoritairement répandue et demeure actuellement le leader dans le domaine du développement Web, surtout au niveau des sites Web WordPress et des applications Web transactionnelles, Node.js, JavaScript et MongoDB se taillent depuis quelques années une part dans le développement d'applications Web intégrant des bases de données non transactionnelles et se positionne au top des sites générant un haut trafic (W3Techs, 2021). Ces technologies servent généralement à générer des services Web qui sont par la suite consumés par les bibliothèques ou cadriciels JavaScript.

1.1.2.2 N-API

N-API est une interface de programmation applicative (API) permettant de développer des extensions (addons) natives en C ou en C++ pouvant par la suite être compilées et exécutées en JavaScript dans la plateforme logicielle Node.js en important l'extension à l'aide de la méthode `require()` (Node.js, 2021d). Cette API constitue la seule interface logicielle stable proposée par la plateforme logicielle Node.js (Node.js, 2021a), ce qui en garantit une utilisation stable au cours des mises à jour majeures de la plateforme.

En utilisant cette API disponible sous Licence MIT, il est donc possible de produire des extensions performantes, puisque le langage C++ offre de plus grandes possibilités au niveau de la gestion de mémoire que le langage JavaScript. L'API permettant de recevoir et d'émettre des objets utilisables en JavaScript sous l'espace de nom NAPI, il est par la suite possible pour un développeur JavaScript d'utiliser l'extension aussi facilement qu'avec le langage JavaScript. La figure 1.4 présente le diagramme schématisant l'architecture Node.js intégrant N-API diffusé lors de la présentation Next Generation N-API : *A Hands-on Workshop* dans le cadre de l'événement Node+JS Interactive en décembre 2019 (Schulhof *et al.*, 2019).

1.1.3 Techniques d'échange de données informatisées

Au début des années 1970, l'échange de données informatisées (EDI), technique remplaçant les échanges physiques de documents entre entreprises par des échanges entre ordinateurs connectés, fait son apparition. En utilisant des messages aux formats standardisés, il est ainsi possible pour deux programmes de communiquer de l'information sans papier et sans intervention humaine.

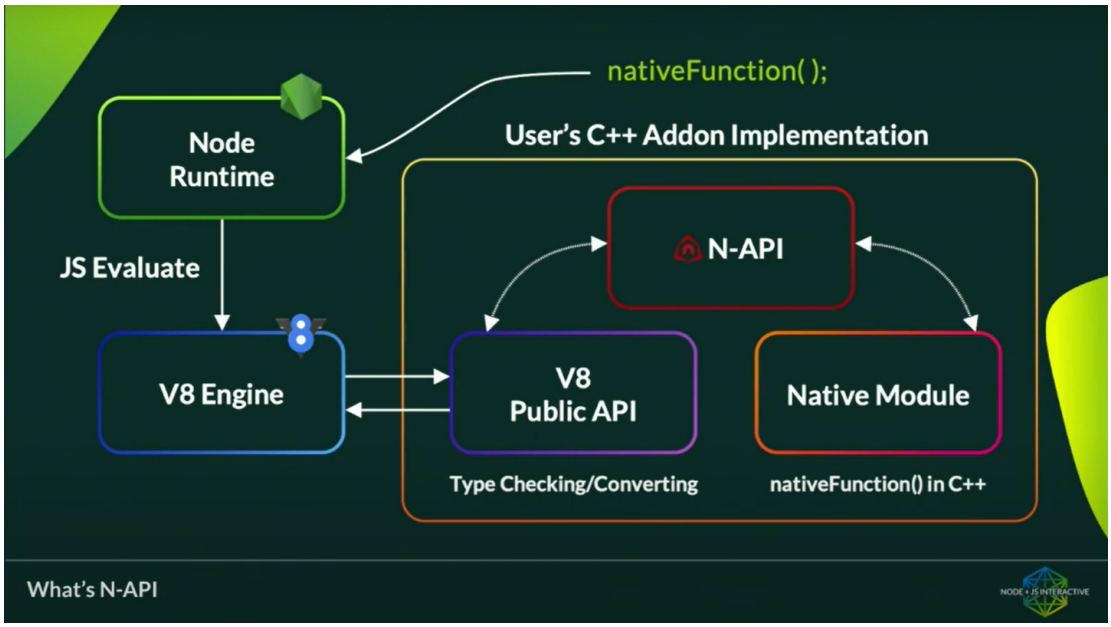

Figure 1.4: Architecture N-API (extraite de la présentation Next Generation N-API : *A hands-on workshop)*

Au début des années 2000, une nouvelle technique d'échange de données fait son apparition, les interfaces de programmation applicatives (API). Bien que les API puissent représenter plusieurs types d'interfaces, dans l'univers du développement Web, les plus connues et utilisées sont les API REST. D'abord utilisée pour partager des données sous forme de Services (SaaS), cette technique est fortement répandue dans le développement d'applications de grande envergure organisées sous forme de microservices, où chacun d'entre eux établit les manipulations possibles avec les données qu'il doit gérer.

1.1.3.1 REST

REST (Representational State Transfer) est un style d'architecture logicielle définissant un ensemble de contraintes à utiliser pour créer des services Web. Les services Web conformes au style d'architecture REST, aussi appelés services Web

RESTful, établissent une interopérabilité entre les ordinateurs sur Internet. Les services Web REST permettent aux systèmes effectuant des requêtes de manipuler des ressources Web via leurs représentations textuelles à travers un ensemble d'opérations uniformes et prédéfinies sans état. Les requêtes effectuées sur l'URI d'une ressource produisent une réponse dont le corps est formaté en HTML, XML, JSON ou un autre format. Lorsque le protocole HTTP est utilisé, comme c'est souvent le cas, les méthodes HTTP disponibles sont GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS et TRACE.

Avec l'utilisation d'un protocole sans état et d'opérations standards, les systèmes REST visent la réactivité, la fiabilité et l'extensibilité, par la réutilisation de composants pouvant être gérés et mis à jour sans affecter le système global, même pendant son fonctionnement (Fielding et Taylor, 2002).

1.1.3.2 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) est un format standard utilisé pour représenter des données structurées de façon semblable aux objets JavaScript popularisé par Douglas Crockford. Il est habituellement utilisé pour structurer et transmettre des données sur des sites Web (par exemple, envoyer des données depuis un serveur vers un client afin de les afficher sur une page Web ou vice versa).

Malgré sa syntaxe très similaire à celle des objets littéraux JavaScript, JSON peut être utilisé indépendamment de ce langage et ainsi, de nombreux autres langages de programmation disposent de fonctionnalités permettant d'analyser la syntaxe du JSON et d'en générer.

Le JSON se présente sous la forme d'une chaîne de caractères, ce qui facilite la transmission des données sur un réseau. Une chaîne de caractères au format JSON a donc besoin d'être convertie en un objet JavaScript natif afin de manipuler ses

données (MDN Web Docs, 2021c). La figure 1.5 représente un objet au format JSON. La syntaxe permet également de représenter un tableau d'objets tel que présenté à la figure 1.6.

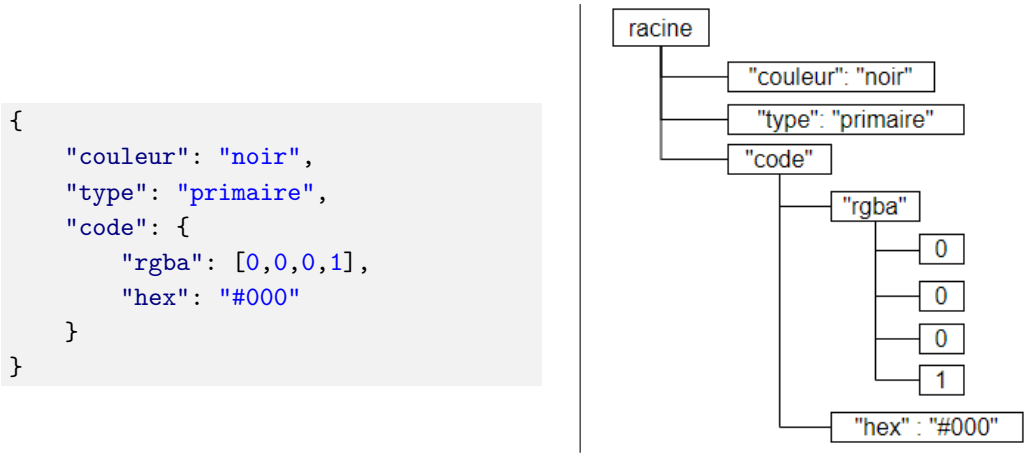

Figure 1.5: Exemple d'objet JSON

Lorsqu'un développeur utilise le langage JavaScript, il est pratique courante de manipuler un objet JSON. Ces manipulations ont en général deux objectifs :

— Générer un document JSON afin de le transmettre à une tierce ressource.

— Recevoir un document JSON afin d'en traiter le contenu.

Ce mémoire porte sur le second volet. Plusieurs manipulations sont possibles en fonction du besoin à combler. Les prochains paragraphes présentent divers contextes où il est nécessaire de manipuler un objet JSON.

```
[
    {
        "couleur": "noir",
        "type": "primaire",
        "code": {
            "rgba": [0,0,0,1],
            "hex": "#000"
        }
    },
    {
        "couleur": "blanc",
        "type": "primaire",
        "code": {
            "rgba": [255,255,255,1],
            "hex": "#FFF"
        }
    }
]
```

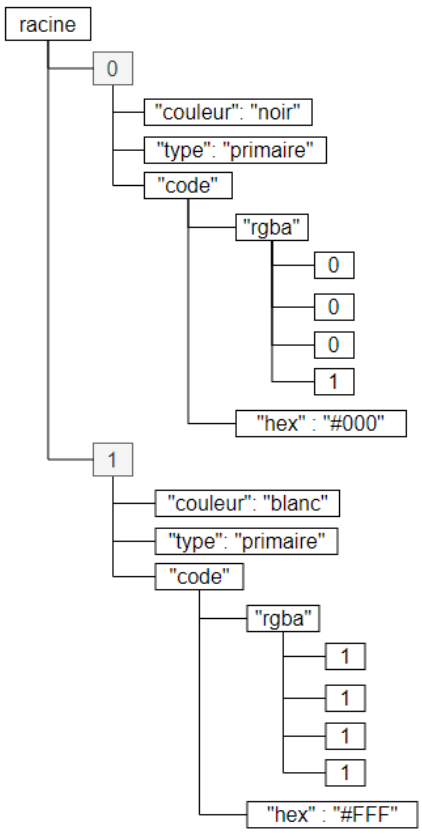

Figure 1.6: Exemple de tableau d'objets JSON

Tout d'abord, il est pratique courante de vouloir accéder à la valeur d'une propriété spécifique. Selon le contexte, la valeur pourra être affichée, faire l'objet d'une opération arithmétique ou logique, ou transformée en fonction du besoin. L'extrait de code 1.1 présente le code JavaScript nécessaire pour accéder à une propriété d'un objet JSON. L'extrait de code 1.2 présente quant à lui le code JavaScript permettant d'accéder à une propriété d'un objet d'un tableau.

```
const objet = {
    "couleur": "noir",
    "type": "primary",
    "code": {
        "rgba": [0,0,0,1],
        "hex": "#000"
    }
}


objet.couleur       // noir
objet.code.rgba     // [0,0,0,1]
objet.code.rgba[3]  // 1
```

Extrait de code 1.1: JavaScript : accès aux propriétés d'un objet

Une seconde pratique courante consiste à obtenir la taille du tableau afin de présenter le nombre de résultats affichés à l'utilisateur en utilisant la propriété `length`. Un exemple d'utilisation de cette propriété est présenté par l'extrait de code 1.3.

Dans un contexte de manipulation de données massives, il est courant que divers objets d'un tableau n'aient pas nécessairement les mêmes propriétés. Dans certaines situations, il s'avère utile d'énumérer la liste des propriétés d'un objet particulier ou de valider la présence d'une propriété en particulier. En JavaScript, ce type d'opération s'effectue à l'aide de la méthode Object.keys(). L'extrait de code 1.4 présente un exemple de manipulation des propriétés.

```
const objets = [
    {
        "couleur": "noir",
        "type": "primary",
        "code": {
            "rgba": [0,0,0,1],
            "hex": "#000"
        }
    },
    {
        "couleur": "blanc",
        "type": "primary",
        "code": {
            "rgba": [255,255,255,1],
            "hex": "#FFF"
        }
    }
]

objets[0]          // Object {couleur: "noir", type: "primary", code: Object}
objets[1].couleur  // blanc
objets[1].code     // Object {rgba: Array(4), hex: '#FFF'}
```

Extrait de code 1.2: JavaScript : accès à une propriété d'un objet dans un tableau

```
objet.length    // Undefined
objets.length   // 2
```

Extrait de code 1.3: JavaScript : obtention de la taille d'un objet ou d'un tableau

```
Object.keys(objet)          // Array(3) ["couleur", "type", "code"]
Object.keys(objets)         // Array(2) ["0", "1"]
Object.keys(objets[1])      // Array(3) ["couleur", "type", "code"]
"couleur" in objets[0]      // true
"saturation" in objets[0]   // false
```

Extrait de code 1.4: JavaScript : obtention des propriétés de divers objets

Le parcours d'un tableau est aussi fréquemment utilisé, que ce soit pour afficher une liste contenant les enregistrements récupérés ou pour traiter le tableau par diverses bibliothèques tierces. La figure 1.5 présente un exemple de parcours d'éléments.

```
for (const objet of objets) {
    objet.couleur       // noir, blanc
}
```

Extrait de code 1.5: JavaScript : Parcours d'un tableau d'objets

Plusieurs autres manipulations peuvent être effectuées sur les objets JSON au format représentant un tableau à l'aide du prototype Array (MDN Web Docs, 2019a). Par exemple, il est possible de trouver l'indice d'un élément à l'aide de la méthode `indexOf()` ou de filtrer les résultats en fonction d'un critère spécifique à l'aide de la méthode `filter()`. Le prototype Array propose également les méthodes `map()` et `reduce()` qui sont fréquemment utilisées dans le traitement de données massives. Plus d'information sur la manipulation d'objets JSON est disponible à l'adresse suivante : `https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON`.

1.2 Techniques d'analyse syntaxique

L'analyse syntaxique consiste à décomposer une phrase ou une chaîne de caractères en segments. Bien que généralement naturelle, cette terminologie est également employée pour l'analyse d'un programme informatique. L'analyseur syntaxique (parser, en anglais) est le programme qui réalise cette tâche. Cette opération suppose une formalisation du texte, qui est vue le plus souvent comme un élément d'un langage formel, défini par un ensemble de règles de syntaxe formant une grammaire formelle. La structure révélée par l'analyse donne alors précisément la façon dont les règles de syntaxe sont combinées dans le texte. Cette structure est

souvent une hiérarchie de syntagmes, représentable par un arbre syntaxique dont les nœuds peuvent être décorés (dotés d'informations complémentaires).

L'analyse syntaxique fait habituellement suite à une analyse lexicale qui découpe le texte en un flux (parfois un graphe orienté acyclique) de lexèmes, et sert à son tour de préalable à une analyse sémantique. Connaître la structure syntaxique d'un énoncé permet d'expliciter les relations de dépendance (par exemple entre sujet et objet) entre les différents lexèmes, puis de construire une représentation du sens de cet énoncé (Wikipédia, 2021).

1.2.1 Méthode JavaScript native JSON.parse()

La méthode `JSON.parse()` offerte par le langage JavaScript analyse syntaxiquement une chaîne de caractères au format standard JSON et construit la valeur JavaScript ou l'objet décrit par cette chaîne (MDN Web Docs, 2019c).

L'extrait de code 1.6 présente un exemple d'utilisation de l'appel de la méthode `JSON.parse()` avec différentes chaînes de caractères représentant des formats JSON de divers types.

```
JSON.parse('{}');              // {}
JSON.parse('true');            // true
JSON.parse('"texte"');         // "texte"
JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]
JSON.parse('null');            // null

var texte = '{ "nom":"John", "ville":"New York"}';

var obj = JSON.parse(texte);    // Object {nom: "John", ville: "New York"}

var nom = obj.nom;              // John
```

Extrait de code 1.6: JavaScript : Analyse syntaxique avec la méthode JSON.parse()

Cette méthode est fréquemment utilisée lors d'échanges de données entre plusieurs

serveurs, notamment lors d'appels à des interfaces de programmation (API) permettant d'obtenir de l'information d'un service tiers. L'extrait de code 1.7 présente un exemple d'appel permettant d'obtenir un tableau de 100 objets représentant les répertoires (repositories) de l'API officielle fournie par GitHub accessible via l'URL `https://api.github.com/repositories`.

```
let xhr = new XMLHttpRequest();

xhr.open('GET', 'https://api.github.com/repositories');

xhr.send();

xhr.onload = function() {
  var repositories = JSON.parse(xhr.response);  // (100) [{...}, {...}, ...]
};
```

Extrait de code 1.7: JavaScript : Analyse syntaxique d'un document provenant d'un serveur externe

Avec l'arrivée de nouveaux bibliothèques et cadriciels (frameworks) JavaScript, l'analyse syntaxique d'un document JSON est désormais intégrée et transparente par l'utilisateur qui, suite à un appel à un service tiers, reçoit un objet prêt pour l'utilisation. Une recherche dans le code source des quatre bibliothèques et cadriciels les plus utilisés, React, Vue.js, Angular et jQuery permet de constater que l'appel à la méthode JSON.parse() est toujours bien présent pour convertir le résultat obtenu par la requête HTTP.

— React : Une recherche des mots-clés "JSON.parse()" dans le code source disponible sur GitHub (Facebook, 2021a) retourne plus d'une trentaine de résultats. Dans la foire aux questions de la documentation officielle React, une section est dédiée aux appels d'API (React, 2021), qui recommande l'utilisation de n'importe quelle bibliothèque AJAX. Le code source contient également un nouveau composant expérimental react-fetch (Facebook, 2021b). Un regard rapide au code source du fichier ReactFetchNode.js

permet de constater que la réponse obtenue est convertie par la méthode `JSON.parse()` lorsque l'objet est de type JSON.

— Vue.js : Dans sa documentation officielle, Vue.js recommande l'utilisation d'Axios pour consommer des API (Vue.js, 2019). Dans ce cas précis, l'appel à une URL externe est effectué à l'aide de la méthode `get()`. Dans le code source de la bibliothèque Axios disponible sur GitHub (Axios, 2021), il est possible d'observer dans le fichier defaults.js l'utilisation de la méthode `JSON.parse()` lors de la transformation de la réponse obtenue.

— Angular : La documentation officielle d'Angular propose l'importation du module HttpClient pour communiquer avec un service tiers (Angular, 2021). Ce module utilise quant à lui la classe HttpXhrBackend qui s'occupe de traiter une requête vers un service externe. Le fichier xhr.ts publié sur la plateforme GitHub permet également d'observer que la méthode `JSON.parse()` est appelée lorsque le type de réponse demandé est `'json'` ou `'text'` (Angular, 2020).

— jQuery : Bien que les tendances actuelles démontrent une diminution de l'utilisation de la bibliothèque jQuery suite à l'apparition des nouvelles bibliothèques et cadriciels JavaScript, son utilisation dans les applications Web existantes ne fait aucun doute. En jQuery, pour faire appel à un service tierce, la méthode `$àjax()` doit être utilisée. En consultant le code source publié sur la plateforme GitHub, il est possible de constater que la méthode `JSON.parse()` est également appelée pour convertir la réponse en objet JavaScript (jQuery, 2021).

Ainsi, advenant qu'il soit possible d'améliorer les performances de cette méthode, c'est une majorité d'applications Web qui pourraient en bénéficier et, au bout du compte, l'utilisateur.

1.2.2 Architecture SIMD et bibliothèque simdjson

Depuis l'invention du tout premier ordinateur, des recherches se poursuivent afin de toujours offrir de meilleures performances et de répondre aux besoins qui sont continuellement grandissants. Pour obtenir des résultats, il importe de constamment repenser les architectures afin de trouver de nouvelles solutions. En 1966, Michael Flynn propose une classification des architectures d'ordinateurs. Nommée taxonomie de Flynn, la classification propose 4 grandes catégories d'architectures d'ordinateur : SISD, SIMD, MISD et MIMD.

1.2.2.1 Architecture SIMD

Single Instruction on Multiple Data (signifiant en anglais : "instruction unique, données multiples"), ou SIMD, est l'une des quatre catégories d'architecture définies par la taxonomie de Flynn. Ce type d'architecture désigne un mode de fonctionnement des ordinateurs dotés de capacités de parallélisme. Dans ce mode, la même instruction est appliquée simultanément à plusieurs données pour produire plusieurs résultats. Le modèle SIMD, présenté à la figure 1.7, convient particulièrement bien aux traitements dont la structure est très régulière, comme c'est le cas pour le calcul matriciel. Généralement, les applications qui profitent des architectures SIMD sont celles qui utilisent beaucoup de tableaux, de matrices, ou de structures de données du même genre. On peut notamment citer les applications scientifiques, ou de traitement du signal (Patterson et Hennessy, 2016).

1.2.2.2 simdjson

La bibliothèque `simdjson` (simdjson, 2021b) proposée par Langdale et Lemire et disponible sur la plateforme GitHub (simdjson, 2021a) sous license Apache 2.0,

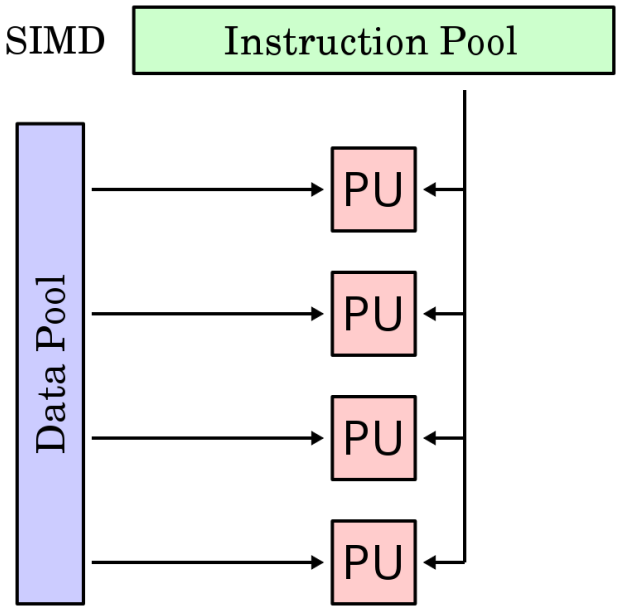

Figure 1.7: Principe du mode SIMD

exploite les possibilités offertes par une architecture SIMD afin d'améliorer la rapidité de traitement de documents au format JSON.

L'article "Parsing Gigabytes of JSON per Second" (Langdale et Lemire, 2019) explique l'architecture de la solution et présente les performances obtenues à grâce à l'algorithme mis en oeuvre. Le graphique de la figure 1.8, extrait de l'article en question, représente les performances obtenues avec divers jeux de données.

La figure 1.9 présente un diagramme de classes sommaire de la structure proposée par la version 0.3.1 publiée en avril 2020. La classe `simdjson::dom::parser` permet de créer un objet qui peut par la suite analyser syntaxiquement un fichier à partir de son chemin d'accès (méthode `load()`) ou à partir d'une chaîne de caractères représentant du texte du format JSON (méthode `parse()`). Par la suite, un document représentant la racine permet d'accéder à l'objet JSON analysé. Il est

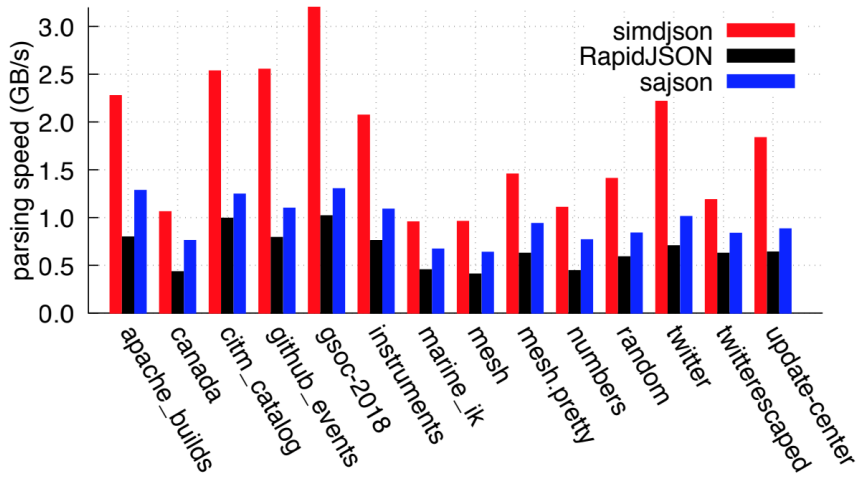

Figure 1.8: Performances de la bibliothèque `simdjson` sur un processeur Skylake (3.4 GHz)

également possible d'obtenir le type du document JSON. Les extraits de code 1.8 et 1.9 présentent respectivement l'analyse syntaxique d'un fichier et d'une chaîne de caractères.

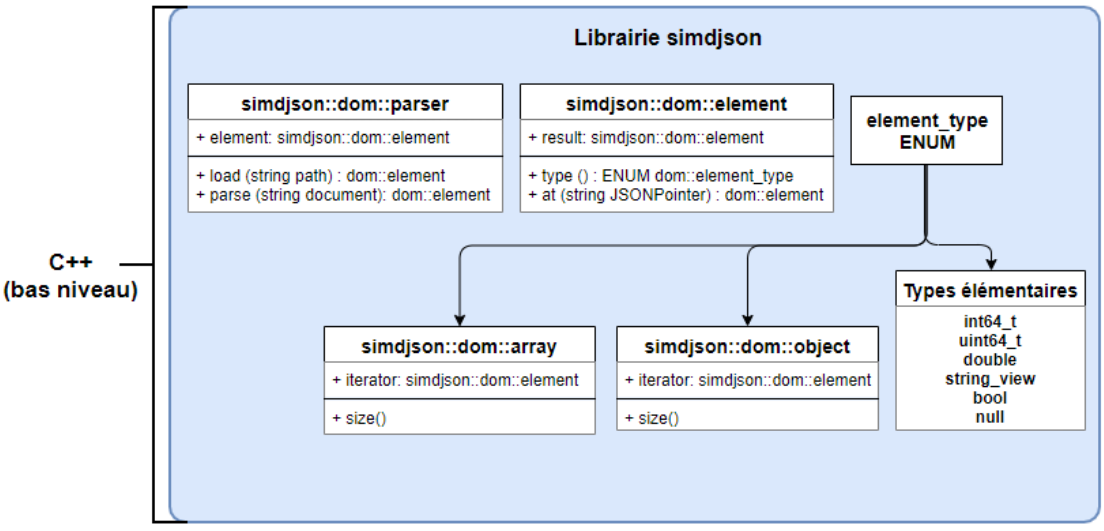

Figure 1.9: Diagramme simdjson

```
dom::parser parser;
dom::element doc = parser.load(filename);
```

Extrait de code 1.8: C++ : Analyse syntaxique d'un fichier au format JSON

```
dom::parser parser;
dom::element doc = parser.parse("[1,2,3]"_padded);
```

Extrait de code 1.9: C++ : Analyse syntaxique d'une chaîne de caractères au format JSON

Les fonctions d'analyse syntaxique `load()` et `parse()` retournent, en fonction de la structure du document analysé, un objet de type `simdjson::dom::element` pouvant par la suite être converti en type élémentaire int64_t, uint64_t, double, string_view, bool ou null, ou sous la forme d'objet défini par la bibliothèque : tableau `simdjson::dom:array` ou objet `simdjson::dom::object`. L'extrait de code 1.10 présente un exemple d'analyse syntaxique et de parcours d'un document JSON structuré en tableau.

```
auto couleurs_json = R"( [
  { "couleur": "noir", "type": "primaire",  "code": { "rgba" : [0, 0, 0, 0,
↪  1], "hex" : "#000"}},
  { "couleur": "blanc", "type": "primaire",  "code": { "rgba" : [255, 255,
↪  255, 255, 255], "hex" : "#FFF"}}
] )"_padded;
dom::parser parser;

// Parcourir un tableau d'objets
for (dom::object objet : parser.parse(couleurs_json)) {
  // Accéder aux propriétés de l'objet
  cout << "couleur/type: " << objet["couleur"] << "/" << objet["type"] <<
  ↪  endl;
}
```

Extrait de code 1.10: C++ : Analyse syntaxique et parcours d'un document JSON structuré en tableau

La bibliothèque `simdjson` implémente également la spécification RFC 6901 - JavaScript Object Notation (JSON) Pointer (Bryan, 2013). Cette spécification définit une syntaxe de chaîne pour identifier une valeur spécifique dans un document JSON (JavaScript Object Notation). L'extrait de code 1.11 présente un exemple de recherche d'élément dans un document JSON structuré en tableau.

```
auto couleurs_json = R"( [
  { "couleur": "noir", "type": "primaire",  "code": { "rgba" : [0, 0, 0, 0,
↪  1], "hex" : "#000"}},
  { "couleur": "blanc", "type": "primaire",  "code": { "rgba" : [255, 255,
↪  255, 255, 255], "hex" : "#FFF"}}
] )"_padded;
dom::parser parser;
dom::element couleurs = parser.parse(couleurs_json);
cout << couleurs_json.at("0/code/rgba/4") << endl; // Affiche 1
}
```

Extrait de code 1.11: C++ : Analyse syntaxique et recherche d'élément selon la spécification RFC 6901 JSON Pointer

La bibliothèque `simdjson` offre également la possibilité d'obtenir le type détecté lors de l'analyse syntaxique d'un document JSON. La possibilité d'obtention du type permet ainsi la mise en oeuvre d'algorithmes permettant de réagir conditionnellement en fonction du type du document. L'extrait de code 1.12 présente un parcours récursif dans le but d'afficher le document analysé dans son intégralité.

Ces exemples ne représentent qu'un aperçu des possibilités d'utilisation de la librairie simdjson. Son développement est constamment mis à jour, dans le but d'offrir un maximum de performances, d'en faciliter l'utilisation et d'offrir plus de fonctionnalités. En date du 25 août 2020, la version la plus récente recommandée pour utilisation grand public est 0.5.0.

1.3 Question de recherche

L'objectif principal des techniques expérimentées dans le cadre de ce mémoire consiste ainsi à conserver le maximum de performances des fonctions d'analyse syntaxiques offertes par la bibliothèque `simdjson` tout en permettant une utilisation conforme aux standards JavaScript.

L'objectif principal peut être décomposé en trois objectifs secondaires :

- Offrir un outil d'analyse syntaxique plus performant que celui nativement offert dans le langage de haut niveau JavaScript sous la plateforme logicielle Node.js.
- Offrir les méthodes standard de traitement d'objets JSON (affichage, dénombrement, énumération et recherche).
- Minimiser les pertes de performances possibles lors du passage de l'objet `simdjson::dom::parser` au langage de haut niveau JavaScript.

Les expérimentations visées par ce mémoire ont été sélectionnées dans le but de répondre à cette question : Est-il possible d'intégrer la bibliothèque `simdjson` dans le langage de haut-niveau JavaScript afin d'analyser syntaxiquement un document, tout en conservant la facilité d'utilisation d'un objet JSON, et en obtenant de meilleures performances que les foncions natives équivalentes ?

1.4 Pertinence économique et institutionnelle

L'utilisation de la bibliothèque `simdjson` peut ajouter un apport significatif au traitement de fichiers volumineux. Dans une ère où l'information est au cœur de la majorité des secteurs d'activités, ces recherches permettent des avancements dans le développement de solutions nécessitant le forage de données, l'intelligence artificielle ainsi que l'intelligence des affaires.

De plus, la bibliothèque `simdjson` présente actuellement un intérêt marqué parmi certains acteurs de l'industrie informatique d'un peu partout dans le monde. À ce jour, le projet compte plus de 12 000 étoiles sur la plateforme GitHub (12 027 en date du 25 août 2020). La bibliothèque figure également dans la liste des ressources modernes et géniales en C++ (Awesome Modern C++) (Rigtorp, 2021), est utilisée par Microsoft FishStore (Microsoft, 2019), Yandex ClickHouse et Clang Build Analyzer.

L'apport de la bibliothèque `simdjson` dans la plateforme logicielle que Node.js, dans un contexte où les objectifs sont rencontrés, offrira un gain de performances dans le traitement de fichiers JSON pour une plus grande proportion de développeurs préférant utiliser un langage de haut niveau tel que JavaScript pour sa facilité d'utilisation.

De plus, considérant que Node.js utilise le moteur JavaScript V8, des résultats concluants pourraient alors présenter un intérêt auprès des preneurs de décisions de ce moteur dans le but d'optimiser la méthode JSON.parse() et peut-être mener à d'autres moteurs JavaScript à envisager l'utilisation de la bibliothèque `simdjson` ou d'autres techniques de parallélisme pour augmenter les performances de ceux-ci.

1.5 Travaux existants

Les expérimentations présentées dans ce mémoire s'inspirent des solutions étudiées jusqu'à présent et tentent de résoudre les points en suspens. Une technique régulièrement utilisée afin de contrecarrer des problèmes de performances lors de chargement de données est de mettre en place le parton de conception "Chargement paresseux", plus communément appelé "Lazy loading". Ce patron de conception propose un chargement dynamique en fonction des besoins de l'utilisateur. Ainsi, au départ, seuls les éléments jugés essentiels sont chargés, alors que les autres sont

chargés sur demande.

Cette technique a été étudiée par Luiz Fernando Peres et comporte ses propres défis. La solution `simdjson_nodejs`, publiée sur la plateforme GitHub (Peres, 2021), propose deux fonctions, `simdjson.parse()` et `simdjson.lazyParse()`, représentées dans le diagramme de classes de la figure 1.10.

1.5.1 Fonction simdjson.parse()

La fonction `simdjson.parse()` appelle la bibliothèque C++ `simdjson` afin de récupérer, à partir du document au format JSON passé en paramètres sous forme de chaîne de caractères, une arborescence contenant le document analysé syntaxiquement. Une fois l'arborescence récupérée, une fonction récursive convertit le tout en objet JavaScript à l'aide du module officiel offert par Node.js, N-API. L'extrait de code 1.13 présente l'exemple d'analyse syntaxique effectuée à l'aide de la méthode `simdjson.parse()` proposé par la documentation officielle de la bibliothèque `simdjson_nodejs` (Peres, 2019).

Des tests de performance effectués sur la fonction ont permis d'identifier que la méthode `makeJSONObject()` utilisée dans l'algorithme proposé pour générer l'objet JavaScript diminue les performances de l'analyse syntaxique, rendant ainsi les performances inférieures à celles de la fonction native `JSON.parse()`. Le problème a été soulevé sur la plateforme GitHub (Lemire, 2019d).

1.5.2 Fonction simdjson.lazyParse()

En réaction aux problèmes de performances de la fonction `simdjson.parse()`, la fonction `simdjson.lazyParse()` a été ajoutée par Luiz Fernando Peres. Cette fonction appelle également la bibliothèque C++ simdjson afin de récupérer, à

partir d'un document passé en paramètre sous forme de chaîne de caractères, une arborescence contenant le contenu du fichier. Par la suite, la fonction retourne un pointeur représenté sous forme de tampon (buffer) ainsi qu'une fonction `valueForKeyPath()`, permettant ainsi de récupérer la valeur d'un item recherché en fonction de sa clé. L'extrait de code 1.14 présente un exemple d'utilisation de ces deux méthodes proposé par la documentation officielle de la bibliothèque `simdjson_nodejs`.

La fonction `simdjson.lazyParse()` offre des performances satisfaisantes. La fonction ne permet cependant pas une utilisation de l'objet dans un format standard (JavaScript Object Notation, JSON), ce qui pourrait représenter une difficulté d'utilisation pour les développeurs familiers avec le format standard. Il n'est pas non plus possible de parcourir l'objet sans accéder explicitement à ses propriétés, par exemple, avec l'instruction for...of ou en utilisant la méthode `Object.keys()`.

```
void print_json(dom::element element) {
  switch (element.type()) {
    case dom::element_type::ARRAY:
      cout << "[";
      for (dom::element child : dom::array(element)) {
        print_json(child);
        cout << ",";
      }
      cout << "]";
      break;
    case dom::element_type::OBJECT:
      cout << "{";
      for (dom::key_value_pair field : dom::object(element)) {
        cout << "\"" << field.key << "\": ";
        print_json(field.value);
      }
      cout << "}";
      break;
    case dom::element_type::INT64:
      cout << int64_t(element) << endl;
      break;
    case dom::element_type::UINT64:
      cout << uint64_t(element) << endl;
      break;
    case dom::element_type::DOUBLE:
      cout << double(element) << endl;
      break;
    case dom::element_type::STRING:
      cout << std::string_view(element) << endl;
      break;
    case dom::element_type::BOOL:
      cout << bool(element) << endl;
      break;
    case dom::element_type::NULL_VALUE:
      cout << "null" << endl;
      break;
  }
}
```

Extrait de code 1.12: C++ : Fonction récursive parcourant l'objet afin d'en afficher le contenu en entier

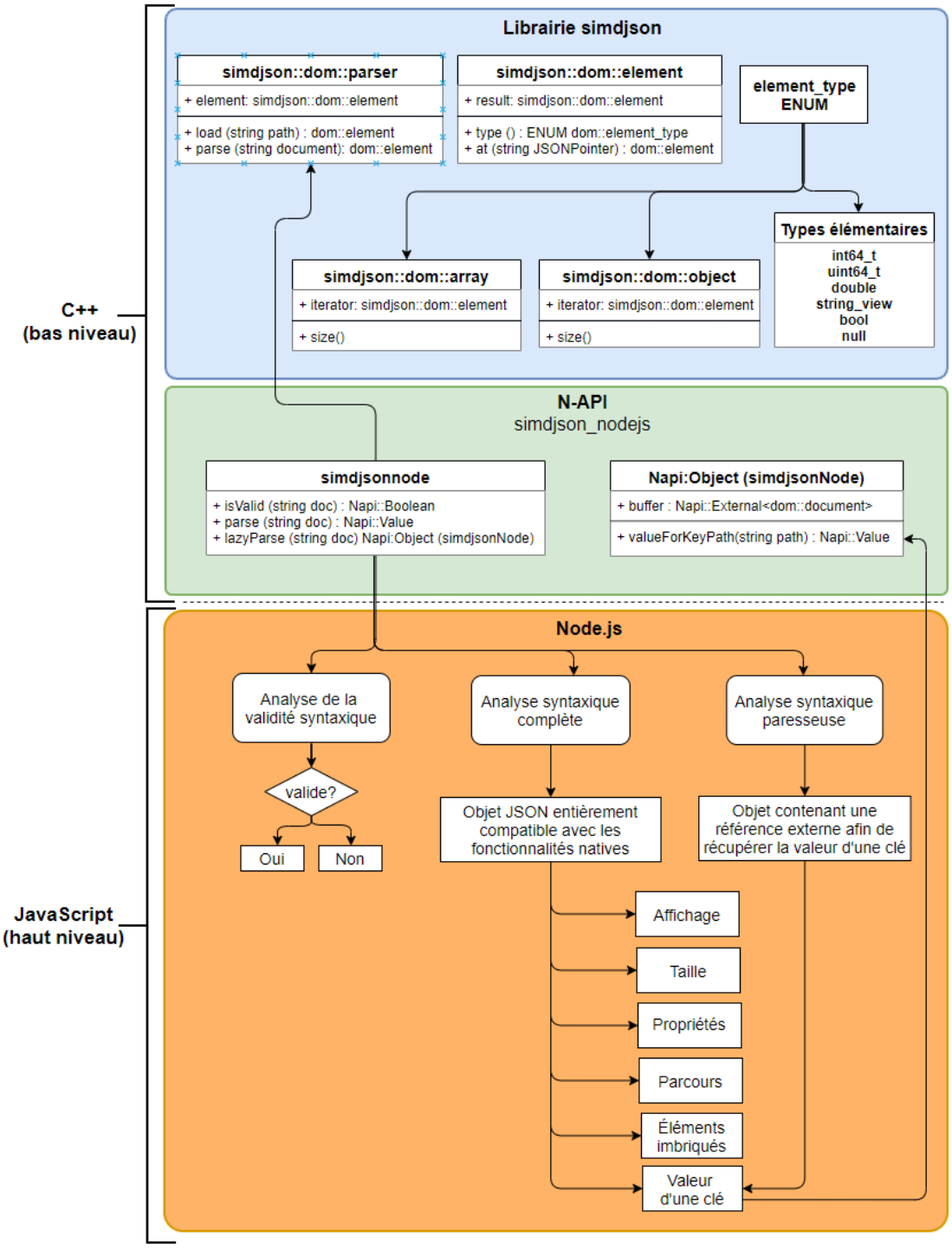

Figure 1.10: Diagramme de la solution `simdjson_nodejs` proposée par Luiz Fernando Peres

```
const simdjson = require('simdjson');

const jsonString = "{
  "foo": {
    "bar": [ 0, 42 ]
  }
}";
const parsedJSON = simdjson.parse(jsonString); // objet JavaScript complet
```

Extrait de code 1.13: JavaScript : Appels de la fonction simdjson.parse() proposée par la bibliothèque `simdjson_nodejs`

```
const simdjson = require('simdjson');

const jsonString = "{
  "foo": {
    "bar": [ 0, 42 ]
  }
}";
const JSONbuffer = simdjson.lazyParse(jsonString);
console.log(JSONbuffer.valueForKeyPath("foo.bar[1]")); // 42
```

Extrait de code 1.14: JavaScript : Analyse syntaxique paresseuse et appel de la fonction valueForKeyPath() proposée par la bibliothèque simdjson_nodejs

CHAPITRE II

MÉTHODOLOGIE

Ce chapitre présente la méthodologie appliquée dans le cadre de ce mémoire. Dans un premier temps, la démarche ayant mené à la sélection des expérimentations est expliquée. Par la suite, les divers fichiers de tests utilisés pour valider les résultats ainsi que les composantes matérielles et logicielles sélectionnées pour évaluer les performances des algorithmes utilisés sont présentés. À la fin de ce chapitre se trouvent les diverses échelles de résultats appliquées afin d'émettre des conclusions sur l'atteinte des objectifs fixés.

2.1 Démarche

La démarche a été divisée en plusieurs étapes, la première portant sur l'étude de la bibliothèque `simdjson` pour en comprendre son fonctionnement. Par la suite, une recherche des divers projets en cours sur l'utilisation de la bibliothèque `simdjson` dans des langages de haut niveau a été effectuée, ce qui a mené à l'étude et l'évaluation du projet `simdjson_nodejs` proposé par Luiperez. Cette étude a permis d'identifier précisément les appels générant des pertes de performances au niveau de la fonction `simdjson.parse()` et d'identifier les limitations de fonctionnalités de la fonction `simdjson.lazyParse()`.

Une première expérimentation a été proposée afin de mesurer l'impact sur les performances et sur les fonctionnalités en utilisant une stratégie visant à offrir un équilibre entre les deux fonctions proposées par la bibliothèque `simdjson_nodejs`. La technique mise en oeuvre consiste à charger le premier niveau des propriétés d'un objet et d'assigner aux propriétés de niveaux supérieurs une méthode d'obtention (accesseur) permettant d'accéder à sa valeur sur demande. Dans le cadre de cette expérimentation, les suppositions suivantes ont été émises :

- Il est possible de définir des méthodes d'obtention (accesseurs) sur un objet JavaScript en utilisant un Proxy Javascript ou le module NAPI.
- La fonction `build_parsed_json()` proposée par la version 0.1.2 de la bibliothèque simdjson est optimale.
- Il est possible d'obtenir un temps raisonnable en parcourant que le premier niveau d'un objet JSON, une fois converti par la bibliothèque `simdjson`.
- Le projet sera déployé sur une architecture de type "instruction unique, données multiples" (SIMD).

Cette expérimentation, présentée en détail au chapitre 3, a permis d'identifier que la stratégie sélectionnée présentait un apport significatif au niveau de la facilité d'utilisation de l'objet suite à l'analyse syntaxique, puisque celui-ci peut être utilisé comme un objet JavaScript natif. L'expérimentation a aussi permis d'identifier certains fichiers où les performances étaient similaires à la fonction `simdjson_nodejs.lazyParse()`. Cependant, dans le cas de certains fichiers, les résultats des performances ne sont pas concluants.

Une seconde stratégie a alors été expérimentée. Celle-ci propose d'encapsuler l'objet et de limiter au maximum la conversion des objets en JavaScript. Dans le cadre de cette expérimentation, les suppositions suivantes ont été émises :

- Il est possible d'encapsuler un objet C++ en utilisant le module N-API et d'assigner des méthodes d'obtention permettant de manipuler l'objet en

fonction des besoins de l'utilisateur.

— Les fonctions `load()` et `parse()` proposées par la version 0.3.1 de la bibliothèque simdjson sont optimales.

— Il est possible de réduire l'utilisation des mécanismes de conversion d'objet C++ en objet JavaScript.

— Il est possible de conserver le résultat obtenu par l'appel d'une fonctionnalité pour éviter de le régénérer lors d'un second appel.

— Le projet sera déployé sur une architecture de type "instruction unique, données multiples" (SIMD).

Cette expérimentation, présentée en détail au chapitre 3, permet d'identifier que cette stratégie est en général plus performante que celle visée par la première expérimentation. Elle permet également d'identifier des limites se concentrant principalement sur les performances des fonctionnalités mises en oeuvre afin de parcourir un objet suite à son analyse syntaxique.

Cela conclut les démarches utilisées dans le cadre de ce mémoire. Un avis portant sur la continuité des recherches sur ce sujet sont présentées dans la conclusion du présent mémoire.

2.2 Méthode utilisée pour mesurer les performances

Plusieurs méthodes peuvent être utilisées pour calculer les performances obtenues par un algorithme. Cette section présente la méthode utilisée dans le cadre des expérimentations présentées dans ce mémoire.

Pour chacune des fonctionnalités où les performances sont jugées essentielles, le temps d'appel de la fonction évaluée a été calculé 1 000 fois afin de mesurer le temps moyen. Cette même stratégie a été appliquée aux fonctions considérées équivalentes afin d'offrir une méthode de calcul comparable. Les temps, calculés en millisecondes

ont tous été arrondis à 3 décimales à l'exception du calcul de la méthode `length` qui présente des résultats arrondis à 4 décimales. Les performances ont été mesurées sur divers fichiers de tests présentés à la section 2.2.1 de ce chapitre.

L'objectif principal étant d'offrir des fonctionnalités d'analyse syntaxique les plus performantes, et considérant que la seconde expérimentation semblait offrir des performances plus intéressantes, les diverses méthodes d'analyse syntaxique mise en oeuvre ont fait l'objet de mesures des performances supplémentaires. Ainsi, afin de valider la stabilité des performances évaluées, l'algorithme servant à calculer le temps moyen de l'analyse syntaxique a été exécuté 100 fois. Ceci a notamment permis d'évaluer les performances minimales, moyennes et maximales, ainsi la marge d'erreur des résultats présentés pour cette fonctionnalité cruciale. Les résultats sont présentés en millisecondes et en gigaoctet par seconde (GB/s) afin offrir une échelle de mesure uniforme parmi les divers fichiers de tests utilisés. De plus, afin de valider si les résultats sont comparables sur plus d'une architecture, cette méthode a été appliquée sur deux architectures différentes présentées à la section 2.2.2 de ce chapitre.

2.2.1 Fichiers de tests

Dans le but de présenter des performances comparables aux travaux existants, les fichiers de tests sélectionnés sont les mêmes que ceux utilisés pour les preuves de performances de la bibliothèque `simdjson_nodejs`. Pour approfondir l'analyse des résultats, les fichiers github_events_100.json et github_events_1000.json ont été ajoutés à la liste des fichiers, ce qui permet une meilleure évaluation des performances sur les fonctionnalités s'appliquant sur des documents au format JSON structurés en tableaux. Le tableau 2.1 présente la liste des fichiers de tests utilisés ainsi qu'une description sommaire de la structure de ceux-ci. Tous les

fichiers représentent un format standard JSON et se terminent par l'extension .json.

Fichier	Taille (Ko)	Description
apache_builds	129	Objet contenant 15 propriétés : assignedLabels (tableau vide), mode (texte), nodeDescription (texte), nodeName (texte, vide), numExecutors (nombre entier), description (texte, hyperlien), jobs (tableau de 875 objets, overallLoad (objet vide), primaryView (objet), quietingDown (booléen), slaveAgentPort (nombre entier), unlabeledLoad (objet vide), useCrumbs (booléen), useSecurity (booléen), views (tableau de 4 objets aux propriétés name (texte) et url (texte, hyperlien).
canada	2 199	Objet contenant 2 propriétés : type (texte) et features (tableau d'objets). La propritété features[0].geometry contient au total 55 563 coordonnées au format [nombre décimal, nombre décimal].
citm_catalog	1 737	Objet contenant 11 propriétés. Chacune des propriétés représente soit un objet vide, un objet simple (hiérarchie à un niveau) ou une liste d'objets de diverses complexités (hiérarchie plusieurs niveaux).

github_events github_events_100 github_events_1000	65	Tableaux de respectivement 30, 100 et 1 000 objets. La structure générale d'un objet comporte 7 propriétés : id (texte), type (texte), created_at (date), actor (objet), repo (objet), public (booléen) et playload (objet).
gsoc-2018	273	Liste de 1 264 objets. La structure générale des objets comporte 6 propriétés : @context (texte, hyperlien), @type (texte), name (texte), description (texte), sponsor (objet) et author (objet).
instruments	224	Objet contenant 9 propriétés : graphstate (null), instruments (tableau de 63 d'objets), message (null), name (texte), orderlist (null), patterns (tableau de 240 objets), pluginstate (null), , samples (tableau de 70 objets) et version (nombre entier).
marine_ik	2 961	Objet contenant 7 propriétés : images, geometries, textures, metadata, materials, object, animations. La propriété geometries[0].data.animations.hierarchy contient un tableau d'objets complexes représentant la plus grande proportion du document.

mesh	707	Objet contenant 8 propriétés : batches (tableau d'objets), morphTargets (objet vide), positions (tableau de 10 800 nombres décimaux), tex0 (tableau de 7 200 nombres décimaux), colors (tableau de 3 600 nombres décimaux), influences (tableau de 3 600 tableaux au format [décimal, entier]), normals (tableau de 10 800 nombres décimaux) et indices (tableau de 33 408 nombres entiers). Le fichier ne contient aucun saut de ligne.
mesh.pretty	1 619	Objet au contenu identique au fichier mesh.json, mais offrant une meilleure lisibilité à l'aide de sauts de ligne et de caractères d'indentation.
numbers	147	Tableau contenant 10 001 nombres décimaux.
random	527	Objet contenant 4 propriétés : id (nombre entier), jsonrpc (texte), total (nombre entier) et result (tableau de 1 000 objets).
twitter	632	Objet de 2 propriétés : statuses (tableau de 100 objets) et search_metadata (objet). Le contenu contient des sauts de lignes et des caractères d'indentation pour en faciliter la lecture.
twitterescaped	550	Objet au contenu identique au fichier twitter.json. Le contenu contient des caractères d'échappement plutôt que des sauts de ligne des caractères d'indentation.
update-center	521	Objet contenant 6 propriétés : connectionCheckUrl (texte, hyperlien), core (objet), id (texte), plugins (objets imbriqués), signature (objet) et updateCenterVersion (nombre entier). Le fichier ne contient aucun saut de ligne ni de caractère d'indentation.

Tableau 2.1: Fichiers de tests utilisés pour mesurer les performances

Certains fichiers ont également été utilisés lors de la validation des résultats des fonctionnalités d'affichage d'un objet, du parcours de tableaux et de l'intégration de la spécification RFC 6901 JSON Pointer. Ces fichiers sont détaillés au tableau 2.2.

Fichier	Taille (Ko)	Description
cars	1	Tableau de 3 objets contenant 4 propriétés : make (texte), model (texte), year (nombre entier) et tire_pressure (tableau de 4 nombres décimaux).
demo	1	Objet contenant 1 propriété : Image, objet imbriqué contenant les propriétés Width (nombre entier), Height (nombre entier), Title (texte), Thumbnail (objet), Animated (booléen) et IDs (tableau d'entiers).
themoviedb	13	Tableau de 20 objets contenant 14 propriétés : popularity (nombre décimal), vote_count (nombre entier), video (booléen), poster_path (texte), id (nombre entier), adult (booléen), backdrop_path (texte), original_language (texte), original_title (texte), genre_ids (tableau d'entiers), title (texte), vote_average (nombre décimal), overview (texte) et release_date (date). Extrait de l'API themoviedb disponible sur `https://api.themoviedb.org/`.

Tableau 2.2: Fichiers de tests utilisés pour valider certaines fonctionnalités

2.2.2 Matériel et logiciels

Les expérimentations présentées dans ce mémoire ont été mises en oeuvre à l'aide d'un ordinateur portable optimisé pour une utilisation professionnelle offrant des performances élevées. Pour bénéficier d'un maximum des performances proposées par la bibliothèque `simdjson`, l'ordinateur doit offrir l'implémentation matérielle de l'architecture SIMD.

2.2.3 Serveurs de tests

Afin d'assurer une stabilité et une comparabilité des performances, celles-ci ont été mesurées sur des serveurs de tests. Le premier serveur, utilisé pour présenter

tous les résultats de performances, possède l'architecture Rome. Les spécifications de l'environnement de tests utilisé sont présentées au tableau 2.3.

Composante	Architecture Rome
Matériel	AMD EPYC Rome 7262 8-Core @ 3.20 GHz
	16 cœurs
	16 Go de RAM
Système d'exploitation	CentOS Linux 7 (Core)
Kernel	Linux 3.10.0-1062.4.3.el7.x86_64 x86_64
GCC	9.3
Docker	v19.03.5

Tableau 2.3: Architecture Rome

Dans certains contextes, afin de valider que les performances obtenues au niveau de l'analyse syntaxique soient similaires sous deux architectures différentes et de comparer les performances sur diverses architectures, un second serveur de tests a été utilisé. Celui-ci repose sur une architecture Skylake-X. Le tableau 2.4 présente les composantes matérielles et logicielles utilisées pour cet environnement.

2.2.4 Architecture Docker

En 2013, Docker a introduit ce qui allait devenir la norme de l'industrie pour les conteneurs. Les conteneurs sont une unité logicielle standardisée qui permet aux développeurs d'isoler leur application de son environnement, résolvant ainsi le mal de tête "ça marche sur ma machine". Pour des millions de développeurs aujourd'hui, Docker est le standard pour créer et partager des applications conteneurisées (Docker, 2021). La figure 2.1 présente l'architecture sommaire présentée par Docker, traduite en français.

Composante	Architecture Skylake-X
Matériel	Intel(R) Xeon(R) W-2104 @ 3.20 GHz
	4 cœurs
	32 Go de RAM
Système d'exploitation	Ubuntu 16.04.6 LTS
Kernel	Linux 4.15.0-99-generic x86_64
GCC	9.3
Docker	v18.09.7

Tableau 2.4: Architecture Skylake-X

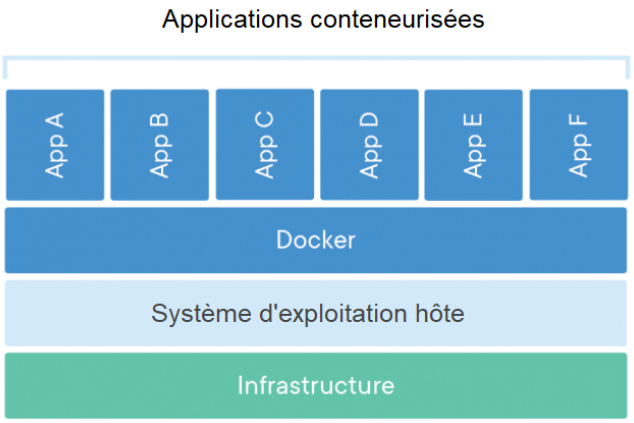

Figure 2.1: Docker

Les expérimentations réalisées dans le cadre de ce mémoire ont été conteneurisées afin de permettre l'évaluation des performances sur les plusieurs serveurs tout évitant d'installer les logiciels nécessaires directement sur les serveurs. Une fois les résultats collectés, les conteneurs utilisés ont été retirés et n'ont ainsi aucun impact sur les serveurs de tests.

2.2.5 Bibliothèques

Les bibliothèques utilisées diffèrent en fonction du contexte des expérimentations et du moment où elles ont été réalisées. Les précisions sont ainsi apportées au niveau du détail de chacune des expérimentations dans le chapitre 3. Pour chacune des expérimentations, une attention particulière a été portée afin que de n'utiliser que les bibliothèques nécessaires pour mettre en oeuvre les stratégies ciblées.

2.3 Résultats attendus

Cette section présente les résultats attendus permettant de mesurer l'atteinte des objectifs fixés par la problématique. Puisque l'objectif principal porte sur la maximisation des performances tout en conservant une fluidité dans l'utilisation des objets générés, plusieurs mesures seront utilisées lors de l'analyse des résultats obtenus par les diverses expérimentations. Ces mesures permettront de répondre aux trois objectifs secondaires préalablement définis.

2.3.1 Performances anticipées

Deux des trois objectifs secondaires identifiés dans la problématique portent sur la maximisation des performances. Les prochaines sections présentent les échelles de mesure utilisées dans ces deux situations. Lors de l'évaluation des performances de l'analyse syntaxique, les performances seront uniformisées à l'unité gigaoctet par seconde pour l'ensemble des fichiers ciblés.

2.3.1.1 Comparaison des performances avec la librairie simdjson

Dans le cas de la bibliothèque `simdjson`, les stratégies expérimentées visent à minimiser les pertes de performances. L'échelle suivante sera utilisée lors de l'analyse des performances obtenues par les diverses expérimentations présentées dans ce mémoire :

- Des performances de moins de 10% inférieures à celles de simdjson seront considérées excellentes ;
- Des performances entre 10% et 25% inférieures à celles de simdjson seront considérées bonnes ;
- Des performances entre 25% et 50% inférieures à celles de simdjson seront considérées moyennes ;
- Des performances de plus de 50% inférieures à celles de simdjson seront considérées non concluantes.

2.3.1.2 Comparaison des performances avec la plateforme logicielle Node.JS

Dans le cas de la méthode d'analyse syntaxique JavaScript `JSON.parse()` offerte par la plateforme Node.js, l'objectif consiste à offrir une méthode d'analyse plus performante que cette dernière. L'échelle suivante sera utilisée lors de l'analyse des performances obtenues par les diverses expérimentations présentées dans ce mémoire :

- Des performances de plus de 75% supérieures seront considérées excellentes ;
- Des performances entre 50% et 75% supérieures seront considérées bonnes ;
- Des performances entre 25% et 50% supérieures seront considérées moyennes ;
- Des performances de moins de 25% supérieures seront considérées non concluantes.

2.3.2 Comparaison de l'utilisation des fonctionnalités offertes

Le troisième objectif secondaire identifié porte sur la capacité d'offrir les méthodes standard de traitement d'objet JSON. L'échelle suivante sera utilisée lors de l'analyse des résultats obtenus par les diverses expérimentations présentées dans ce mémoire :

- Une solution permettant l'utilisation d'un objet 100% compatible avec les méthodes nativement offertes en JavaScript et présentant des résultats identiques sera considérée excellente ;
- Une solution offrant les méthodes standard ou équivalentes de traitement d'objets JSON (affichage, dénombrement, énumération et recherche) et présentant des différences négligeables (ne portant pas sur le contenu, sur la précision des résultats et n'affectant pas l'interprétation des résultats par l'utilisateur) sera considérée bonne ;
- Une solution n'offrant pas les méthodes standard ou équivalentes dans le traitement d'objets JSON (affichage, dénombrement, énumération et recherche) ou présentant des différences notables (portant sur le contenu, la précision des données, ou affectant l'interprétation des résultats par l'utilisateur) sera considérée non concluante.

CHAPITRE III

EXPÉRIMENTATION

3.1 Chargement du premier niveau

La fonction `simdjson.lazyParse()` proposée dans les travaux existants offre des performances satisfaisantes. Cependant, la fonction ne permet pas une utilisation de l'objet dans un format standard (JavaScript Object Notation, JSON), ce qui pourrait représenter une difficulté d'utilisation pour les développeurs initiés à ce format. Il n'est pas non plus possible de parcourir l'objet sans accéder explicitement à ses propriétés, par exemple, avec une boucle de type for...of ou en utilisant la méthode `Object.keys()`.

Cette première expérimentation vise à mettre en oeuvre une solution permettant à la fois un chargement paresseux et l'accès aux propriétés dans le format standard JSON en implémentant une nouvelle fonction, `simdjson.lazyParse1Depth()`, qui tentera d'offrir un équilibre entre les méthodes `simdjson.parse()` et `simdjson.lazyParse()`.

3.1.1 Technique utilisée

La fonction `simdjson.lazyParse1Depth()` prend un document au format JSON en paramètre. Elle en vérifie la validité à l'aide de la fonction `is_valid` de la

bibliothèque simdjson. Le document est ensuite analysé syntaxiquement à l'aide de la fonction `build_parsed_json` de la bibliothèque simdjson.

Par la suite, l'algorithme parcourt le premier niveau de l'objet retourné. Dans le cas de types primitifs (nombre, chaîne de caractères, valeur booléenne, etc.), la valeur est simplement retournée au format JavaScript. Dans le cas d'objets, les propriétés de premier niveau de l'objet sont parcourues. Pour chaque propriété, la méthode `getProperty()` est appelée. Les propriétés primitives sont ajoutées à l'objet en tant que valeurs énumérables. Les propriétés complexes (objets et tableaux) sont associées à des méthodes d'obtention (accesseurs) permettant de récupérer leur contenu dynamiquement.Dans le cas de tableaux, chaque élément du tableau est parcouru. Les tableaux contenant des valeurs primitives sont ajoutés directement à l'objet retourné. Les tableaux contenant des objets complexes (tableaux d'objets ou tableaux de tableaux) sont associés à des méthodes d'obtention (accesseurs) permettant de récupérer leur contenu dynamiquement.

Cette technique permet une utilisation ultérieure possible de l'objet retourné à l'aide des fonctionnalités natives JavaScript implémentées par la plateforme logicielle Node.js. La figure 3.1 présente le diagramme de flux de la solution expérimentée.

Le module intermédiaire développé avec l'interface applicative N-API accède aux fonctions d'analyse syntaxique de la bibliothèque simdjson. Il retourne par la suite un objet de type `Napi::Object` qui est reconverti par l'interface N-API en objet JavaScript. Cela permet d'exploiter les fonctionnalités natives suivantes :

- Affichage ;
- Taille ;
- Propriétés ;
- Parcours.

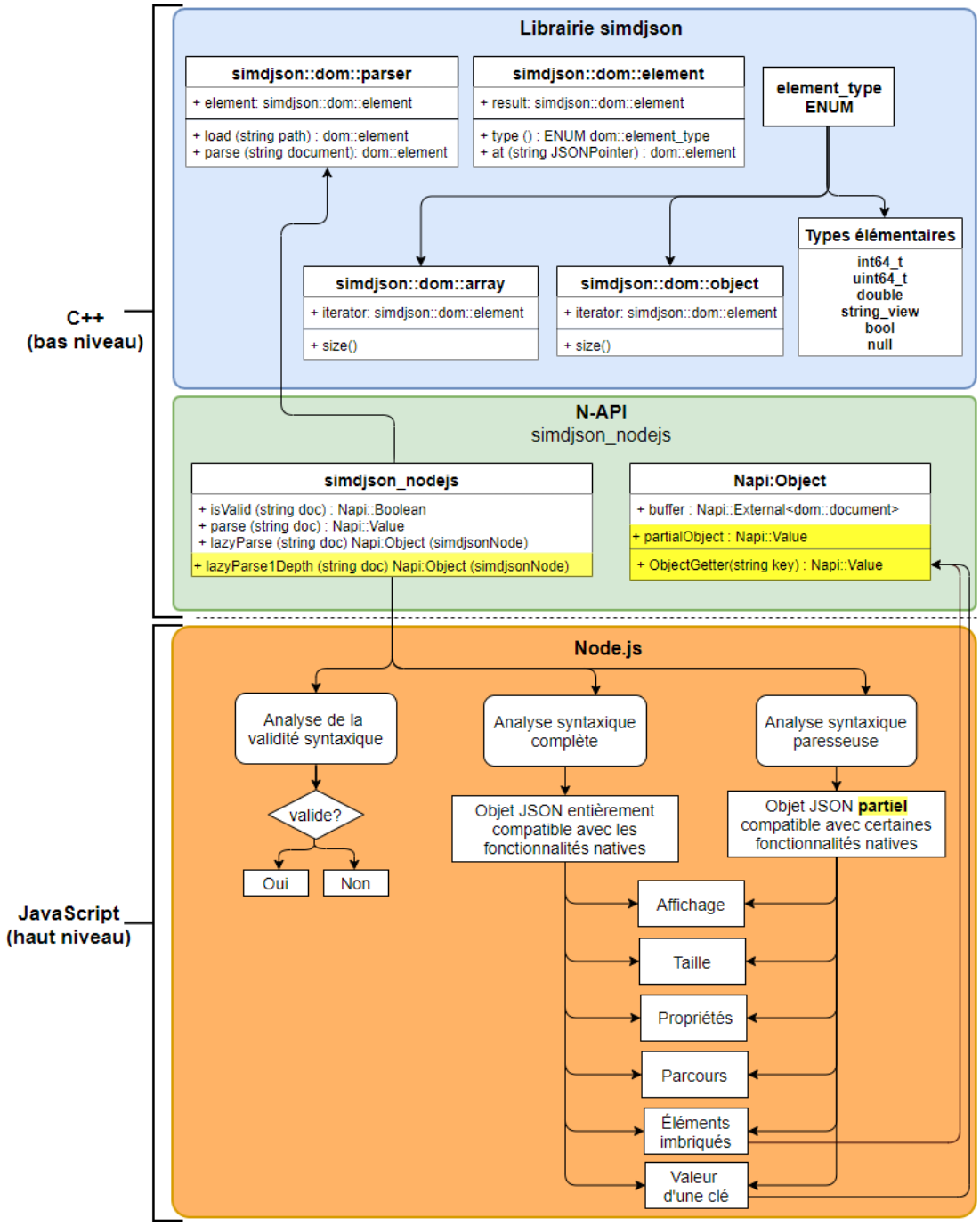

Figure 3.1: Diagramme de flux représentant la technique de chargement du premier niveau

Lors de la manipulation d'un objet imbriqué (de niveau 2 ou plus), la propriété est associée à une méthode d'obtention définie dans le module à l'aide de la méthode ObjectGetter() qui prend la clé sous forme de chaîne de caractères en paramètre. Il est ainsi possible de retourner la valeur de manière transparente pour l'utilisateur.

3.1.2 Infrastructure visée par l'expérimentation

- C++ :
  - simdjson : 0.1.2
- JavaScript (modules npm) :
  - node-addon-api : 1.6.3

La version précédente du module node-addon-api (1.6.2) comporte des problèmes de gestion de mémoire documentés, qui ont été corrigés dans la version suivante (1.6.3) (Node.js, 2021b). La version 1.7.0 publiée le 23 juillet 2019 n'a pas été testée lors de cette expérimentation.

L'architecture Rome a été utilisée pour évaluer les performances présentées dans ce mémoire. Les configurations matérielles et logicielles de cette architecture sont définies par le cadre méthodologique présenté au chapitre 2.

3.1.3 Résultats obtenus par l'expérimentation

La comparaison des objets retournés par la fonction `simdjson.lazyParse1Depth()` et son équivalent natif démontre des résultats identiques pour les fichiers de données de tests représentant une hiérarchie à un seul niveau. Les fonctions natives de traitement d'objets et de tableaux utilisées sur l'objet retourné par la fonction `simdjson.lazyParse1Depth()` sont fonctionnelles et retournent des résultats similaires à un objet JSON natif.

Voici les différences observées entre les 2 objets lors de l'affichage de l'objet à l'aide de l'instruction `console.log()` :

- La valeur d'une propriété imbriquée de l'objet indique `[Getter]` pour la fonction expérimentée `simdjson.lazyParse1Depth()` et `[Object]` pour l'objet JSON natif.
- Des différences de précisions de certains nombres décimaux (par exemple, 0.23033292890999998 plutôt que 0.23033292891 dans le fichier de tests numbers.json) ont également été observées dans la version simdjson utilisée pour réaliser cette expérimentation. Il est important de noter que cette problématique, identifiée dans la bibliothèque `simdjson`, a été documentée (Lemire, 2019b) et corrigée dans une mise à jour survenue le 15 juillet 2019 (Lemire, 2019c). Elle s'explique par le fait que les premières versions de la bibliothèque garantissaient la précision de traitement des nombres à virgule flottante avec une précision correspondant à une *unité dans la dernière place* ; la précision des versions subséquentes est maintenant parfaite.

Afin d'évaluer les performances, 16 fichiers de tests de format JSON valides ont été utilisés. Les performances de la fonction `simdjson.lazyParse1Depth()` ont été comparées aux fonctionnalités `simdjson.parse()` et `simdjson.lazyParse()` déjà mis en place par les travaux existants, ainsi que par l'équivalent natif `JSON.parse()`. Le tableau 3.1 présente les résultats obtenus normalisés sous l'unité "Gigaoctets par secondes (GB/s)". Les performances moyennes sont également présentées sous forme de graphique à la figure 3.2 afin d'offrir une meilleure comparaison visuelle entre les diverses méthodes évaluées.

Le graphique montre que l'algorithme développé n'offre pas les performances escomptées lors de l'analyse syntaxique du fichier gsoc-2018.json. Il est également possible d'observer que la vitesse de la fonction `simdjson.lazyParse1Depth()` est supérieure à celle de la fonction native `JSON.parse()` pour 11 des 16 fichiers

Fichier	JSON parse	simdjson parse	simdjson lazyParse	simdjson lazyParse1Depth
apache_builds.json	0,18	0,06	0,77	0,58
canada.json	0,08	0,04	0,64	0,62
citm_catalog.json	*0,29*	0,07	0,46	0,45
github_events.json	0,23	0,06	0,66	0,08
github_events_100.json	0,25	0,08	0,45	0,16
github_events_1000.json	0,25	0,07	0,45	0,14
gsoc-2018.json	**0,37**	**0,21**	**1,51**	0,20
instruments.json	0,28	0,05	0,71	*0,72*
marine_ik.json	0,14	0,04	0,56	0,55
mesh.json	0,15	0,05	0,62	0,58
mesh.pretty.json	0,26	*0,09*	*0,91*	**0,83**
numbers.json	0,18	0,07	0,56	0,06
random.json	0,14	0,03	0,29	0,31
twitter.json	0,25	0,05	0,38	0,39
twitterescaped.json	0,20	0,05	0,63	0,61
update-center.json	0,16	0,04	0,38	0,38

Tableau 3.1: Performances, en gigaoctet par seconde (GB/s), de l'appel des diverses méthodes d'analyse syntaxique

testés et que les vitesses sont comparables (moins de 0,1 gigaoctet par seconde) à la fonction `simdjson.lazyParse()` pour 10 des 16 fichiers.

Afin d'évaluer le degré d'atteinte des résultats attendus, la variance de la vitesse (en %) en comparaison à la fonction native `JSON.parse()` ainsi que la variation de

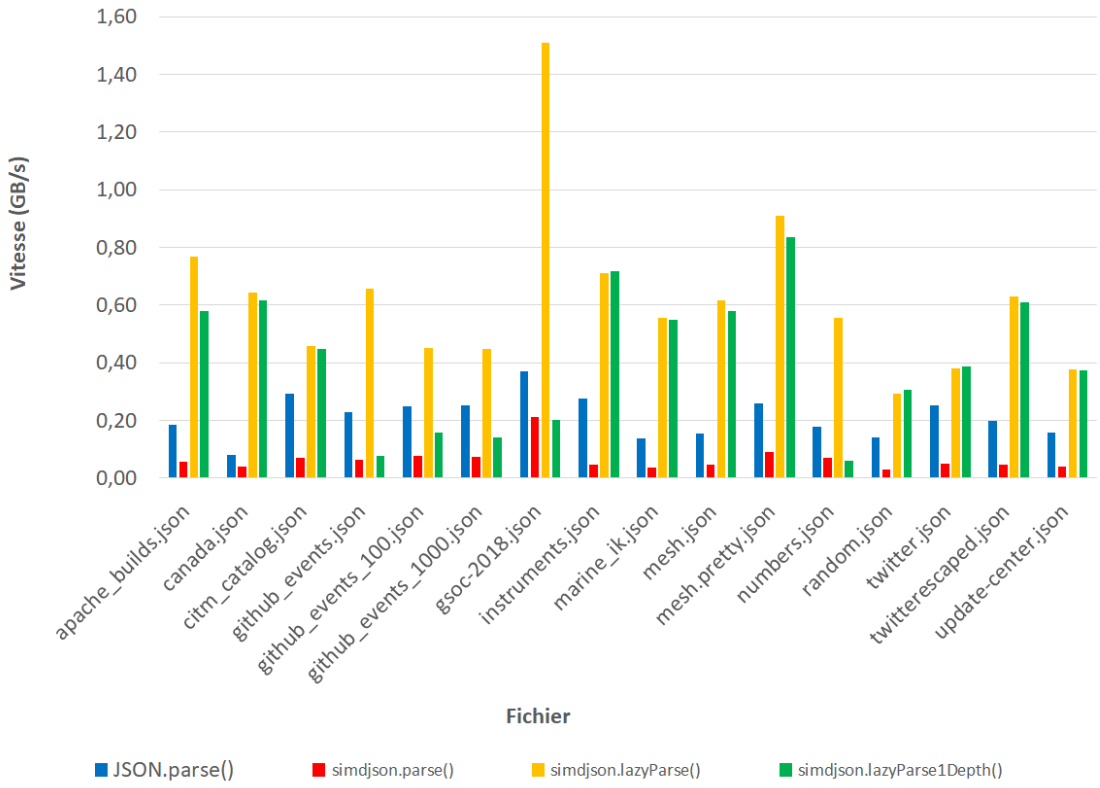

Figure 3.2: Performances, en gigaoctet par seconde (GB/s), de l'appel des diverses méthodes d'analyse syntaxique obtenues par la première expérimentation

la vitesse en comparaison à la fonction `simdjson.lazyParse()` ont été calculées. Ces résultats sont présentés dans les tableaux 3.2 et 3.3.

Les fichiers testés démontrent une augmentation des performances supérieure à 75% pour l'utilisation de la fonction `simdjson.lazyParse1Depth()` en comparaison avec la fonction native `JSON.parse()` pour 9 des 16 fichiers évalués. Ces résultats répondent à l'objectif initial et démontrent d'excellentes vitesses. De bonnes performances (50–75%) ont été observées pour 2 des 16 fichiers, alors que les 5 fichiers restants ont obtenu des performances inférieures à la fonction native.

Les variations de performances entre les fonctions `simdjson.lazyParse()` et

Fichier	JSON parse	simdjson parse	simdjson lazyParse	simdjson lazyParse1Depth
apache_builds.json	0,18	- 69%	+ *317%*	+ 213%
canada.json	0,08	- 50%	+ **711%**	+ **679%**
citm_catalog.json	0,29	- 40%	+ 56%	+ 53%
github_events.json	0,23	- 73%	+ 186%	- 66%
github_events_100.json	0,25	- 81%	+ 126%	- 37%
github_events_1000.json	0,25	- 71%	+ 78%	- 44%
gsoc-2018.json	0,37	- 42%	+ 307%	- 46%
instruments.json	0,28	- 83%	+ 158%	+ 159%
marine_ik.json	0,14	- 74%	+ 305%	+ *300%*
mesh.json	0,15	- 69%	+ 299%	+ 277%
mesh.pretty.json	0,26	- 65%	+ 251%	+ 223%
numbers.json	0,18	- 61%	+ 213%	- 65%
random.json	0,14	- 79%	+ 105%	+ 115%
twitter.json	0,25	- 80%	+ 51%	+ 53%
twitterescaped.json	0,20	- 76%	+ 216%	+ 207%
update-center.json	0,16	- 75%	+ 136%	+ 137%

Tableau 3.2: Variation des performances entre la fonction native `JSON.parse()` et les diverses techniques d'analyse syntaxique évaluées par la première expérimentation

`simdjson.lazyParse1Depth()` démontrent une baisse des performances de plus de 50% pour 5 des 16 fichiers évalués, une baisse de 25% pour l'un d'eux ainsi qu'une baisse se situant entre 1% et 8% pour 6 fichiers.

Il est intéressant de noter que la fonction `simdjson.lazyParse1Depth()` offre dans certains cas des performances égales ou supérieures à la fonction existante

Fichier	simdjson lazyParse()	simdjson lazyParse1Depth()
apache_builds.json	0,77	- 25%
canada.json	0,64	- 4%
citm_catalog.json	0,46	- 2%
github_events.json	0,66	- 88%
github_events_100.json	0,45	- 65%
github_events_1000.json	0,45	- 69%
gsoc-2018.json	1,51	- 87%
instruments.json	0,71	+ 1%
marine_ik.json	0,46	- 1%
mesh.json	0,62	- 6%
mesh.pretty.json	0,91	- 8%
numbers.json	0,56	- 89%
random.json	0,29	+ **4%**
twitter.json	0,38	+ *2%*
twitterescaped.json	0,63	- 3%
update-center.json	0,38	0%

Tableau 3.3: Variation des performances entre la fonction `simdjson.lazyParse()` et la fonction `simdjson.lazyParse1Depth()`

`simdjson.lazyParse()`. Ce résultat s'observe en effet pour 4 des 16 fichiers évalués :

— instruments.json ;

— random.json ;

— twitter.json ;

— update-center.json.

Dans ces 4 cas, les structures proposées ont toutes un point commun : leur hiérarchie de premier niveau contient peu de propriétés et les valeurs associées à certaines d'entre elles correspondent à des objets complexes ou des tableaux représentant un poids significatif du document. Cette observation permet de conclure que l'envoi de l'objet en référence externe peut parfois être plus coûteux en temps que la génération du premier niveau. Il est ainsi donc possible d'affirmer que pour les documents JSON représentant des petits tableaux d'objets complexes, il serait préférable d'utiliser la fonction `simdjson.lazyParse1Depth()`. En plus d'être plus rapide, cette solution offre en plus la possibilité d'obtenir la taille du tableau et la liste des propriétés et d'en parcourir les éléments en utilisant les fonctions natives intégrées.

3.1.4 Restrictions et limitations

L'hypothèse sur la possibilité d'offrir des performances supérieures à celle de la fonction native `JSON.parse()` tout en conservant une utilisation standard de l'objet à l'aide du module node-addon-api répond en partie aux objectifs fixés par cette première expérimentation. Les objets générés peuvent être manipulés de la même manière qu'un objet standard respectant la norme JSON (JavaScript Object Notation). Les travaux réalisés dans cette expérimentation ont également permis de se familiariser avec les modules standards offerts par la plateforme logicielle Node.js pour la mise en oeuvre de modules natifs en utilisant l'interface application N-API. Ils ont également permis de relever des problèmes de gestion de mémoire de la version 1.6.2 du module N-API.

L'algorithme proposé a cependant des limitations. Lors du traitement de fichiers contenant des listes importantes d'objets complexes, la vitesse d'exécution est inférieure à toutes les autres solutions d'analyse syntaxique testées. Il est ainsi

possible d'affirmer que la conversion d'un objet complexe de la bibliothèque C++ à un objet JavaScript est coûteuse en ressources et que d'autres solutions devront probablement être combinées afin d'en améliorer les performances. Les tests effectués ont également permis d'observer des erreurs de traitement dans les fichiers contenant des tableaux d'objets. Ces erreurs devront être corrigées avant d'intégrer cette solution dans une publication officielle.

Tenant compte des résultats observés, nous considérons que cet algorithme mérite d'être combiné à d'autres solutions afin d'augmenter la vitesse d'exécution lors de l'analyse syntaxique de documents au format JSON. La prochaine expérimentation tentera de répondre aux limitations identifiées.

3.2 Encapsulation totale de l'objet dans un langage de bas niveau

Une nouvelle stratégie a ainsi été élaborée afin d'éviter au maximum la conversion des objets de la bibliothèque C++ à un objet JavaScript. Plutôt que d'offrir une seule méthode et de retourner un objet, la stratégie d'encapsulation d'un objet a été mise en oeuvre afin de valider si une telle stratégie peut offrir de meilleures performances tout en conservant la facilité d'utilisation de l'objet en offrant des fonctionnalités similaires à celles pouvant être utilisées avec un objet JSON natif.

3.2.1 Technique utilisée

Le module mis en oeuvre dans cette expérimentation offre à l'utilisateur un objet nommé `simdjsonJS` héritant de la classe `Napi::ObjectWrap` qui définit des propriétés et des méthodes pouvant être utilisée en JavaScript, tout en conservant une partie de la logique de l'objet à l'intérieur du module C++ développé. Ainsi, lorsque l'utilisateur crée un nouvel objet `simdjsonJS` en JavaScript, le constructeur analyse syntaxiquement le document à l'aide des fonctions de la bibliothèque

simdjson. Lorsque la structure du document représente des données organisées sous forme de tableau, la taille est également définie. Une fois cette étape complétée, le constructeur conserve à l'intérieur du module un objet de type `simdjson::dom::parser` pouvant être réutilisé lors des divers appels aux méthodes proposées par le module. Le constructeur retourne un objet contenant les méthodes d'obtentions ainsi que les propriétés implémentées. Dans le cadre de cette expérimentation, les fonctionnalités implémentées et testées sont les suivantes :

- conversion du document en chaîne de caractères (toString), qui permet notamment l'utilisation de la commande `console.log()` sur l'objet afin d'en afficher le contenu ;
- obtention de la taille (nombre d'éléments) du document lorsque le document représente une structure organisée en tableau ;
- obtention de la liste des propriétés d'un objet ou d'un tableau ;
- parcours du document ;
- obtention des éléments imbriqués ;
- recherche selon la spécification JSON Pointer (RFC 6901).

À l'exception de la taille qui est définie par une propriété et ainsi directement accessible une fois l'objet construit, chacune des fonctionnalités offertes est traitée par l'appel d'une méthode d'obtention développée dans le module intermédiaire. Cela évite de construire l'intégralité de l'objet lors de son initialisation et de fournir les résultats requis par l'utilisateur sur demande. De plus, lorsque possible, le résultat généré par l'appel d'une méthode est conservé lors de son premier appel afin d'optimiser les performances des appels subséquents. Les résultats de l'expérimentation permettront d'identifier si cette stratégie offre de meilleures performances au niveau de l'analyse syntaxique, tout en conservant de bonnes performances pour chacune des fonctionnalités offertes. La figure 3.3 présente le diagramme de flux de la solution expérimentée. Le code source développé dans le cadre de cette expérimentation est disponible sur la plateforme GitHub à l'adresse

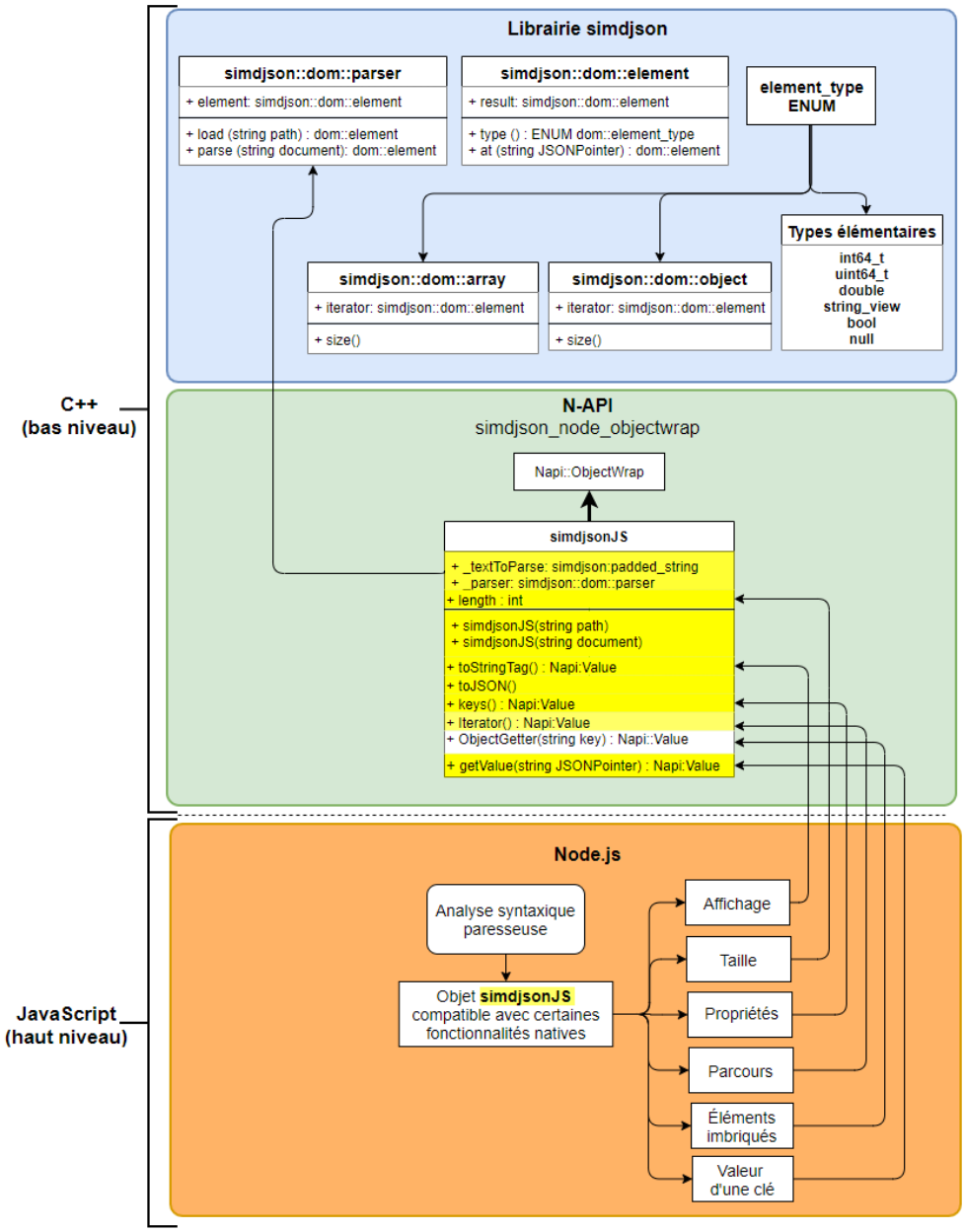

Figure 3.3: Diagramme de flux représentant la technique d'encapsulation de l'objet

suivante : `https://github.com/croteaucarine/simdjson_node_objectwrap`.

3.2.2 Infrastructure visée par l'expérimentation

Tout ordinateur muni d'un système d'exploitation Linux, macOS ou Windows compatible avec les versions logicielles expérimentées sera en mesure d'exécuter le code fourni par cette expérimentation. Un processeur muni d'une architecture permettant l'exécution d'instructions de type "Single instruction multiple data" (SIMD) offrira de meilleures performances.

Dans le cadre de cette expérimentation, les bibliothèques C++ et JavaScript suivantes ont été utilisées :

- C++ :
  - simdjson : 0.3.1
- JavaScript (modules npm) :
  - bindings : 1.5.0
  - json-pointer : 0.6.0
  - node-addon-api : 3.0.0

Les versions du module node-addon-api antérieures à 1.6.3 comportent des problèmes de gestion de mémoire sur certaines des fonctionnalités implémentées et sont donc incompatibles avec la solution proposée. Le correctif est documenté dans l'historique des versions du module (Node.js, 2021b).

Les architectures Rome et Skylake-X ont été utilisées pour mesurer les performances de l'analyse syntaxique présentées dans ce mémoire. Les performances des autres fonctionnalités expérimentées ont été mesurées à l'aide de l'architecture Rome. Le configurations matérielles et logicielles de ces deux architectures sont définies par le cadre méthodologique présenté au chapitre 2.

3.2.3 Résultats obtenus par l'expérimentation

Pour l'utilisateur, l'interface propose plusieurs fonctionnalités lui permettant d'utiliser le document de façon similaire aux fonctions natives JavaScript sur un objet JSON. L'utilisateur peut :

- Procéder à l'analyse syntaxique d'un document ;
- Afficher le contenu de l'objet ;
- Obtenir le nombre d'éléments (taille) d'un document de type tableau ;
- Obtenir la liste des propriétés d'un document de type objet ou tableau ;
- Parcourir l'ensemble d'un objet de type tableau à l'aide d'une boucle for...of ;
- Effectuer une recherche par requête en fonction du pointeur JSON défini par la spécification RFC 6901.

La comparaison des résultats obtenus permet de constater que le format généré par ces opérations est similaire aux opérations nativement offertes. Les performances obtenues permettent d'établir divers constats en fonction de la fonctionnalité évaluée. Une analyse plus détaillée des résultats pour chacune des fonctionnalités implémentée par la solution est présentée dans les prochaines sections.

3.2.3.1 Analyse syntaxique

Tel que défini par la problématique, le premier objectif de cette recherche est de valider s'il est possible d'obtenir des performances significativement supérieures aux opérations nativement offertes dans le langage de haut-niveau JavaScript sous la plateforme logicielle Node.js, en ce qui concerne l'analyse syntaxique d'un document au format JSON. Avant de poursuivre les travaux d'implémentation des diverses méthodes permettant le traitement de l'objet syntaxiquement analysé, ces performances doivent être mesurées et observées.

Les 4 méthodes suivantes ont été expérimentées afin de permettre l'analyse des performances obtenues :

- Méthode 1 : Calcul du temps requis lorsque le document est préalablement chargé en mémoire.
- Méthode 2 : Calcul du temps requis en passant le chemin d'accès au fichier en paramètre.
- Méthode 3 : Calcul du temps requis en passant le document préchargé en paramètre sous forme de chaîne de caractères.
- Méthode 4 : Calcul du temps requis en utilisant la fonction native `JSON.parse()` proposée par JavaScript.

Afin de comparer les performances sous deux architectures différentes, d'analyser la stabilité des diverses méthodes d'analyse syntaxique implémentées et d'identifier la marge d'erreur des résultats présentés, les fonctions d'analyse syntaxique ont été mises à l'épreuve en réalisant des tests de performance. Les scripts de bancs d'essai ont ainsi été exécutés 100 fois et les valeurs minimales, moyennes maximales ont été consolidées. Ces résultats sont présentés graphiquement par les figures 3.4 et 3.5.

Les graphiques démontrent que les performances sont plutôt stables. La marge d'erreur calculée selon un degré de confiance de 99% est inférieure à 1% pour l'ensemble des méthodes évaluées, et ce, dans les 2 environnements testés. Il est également possible d'observer que les performances sont généralement supérieures lors de l'utilisation de l'analyse syntaxique avec la bibliothèque simdjson. Il est aussi possible de constater que l'architecture Rome offre des résultats plus performants que l'architecture Skylake-X, mais que dans l'ensemble, il est possible d'obtenir des conclusions similaires en comparant les diverses méthodes évaluées.

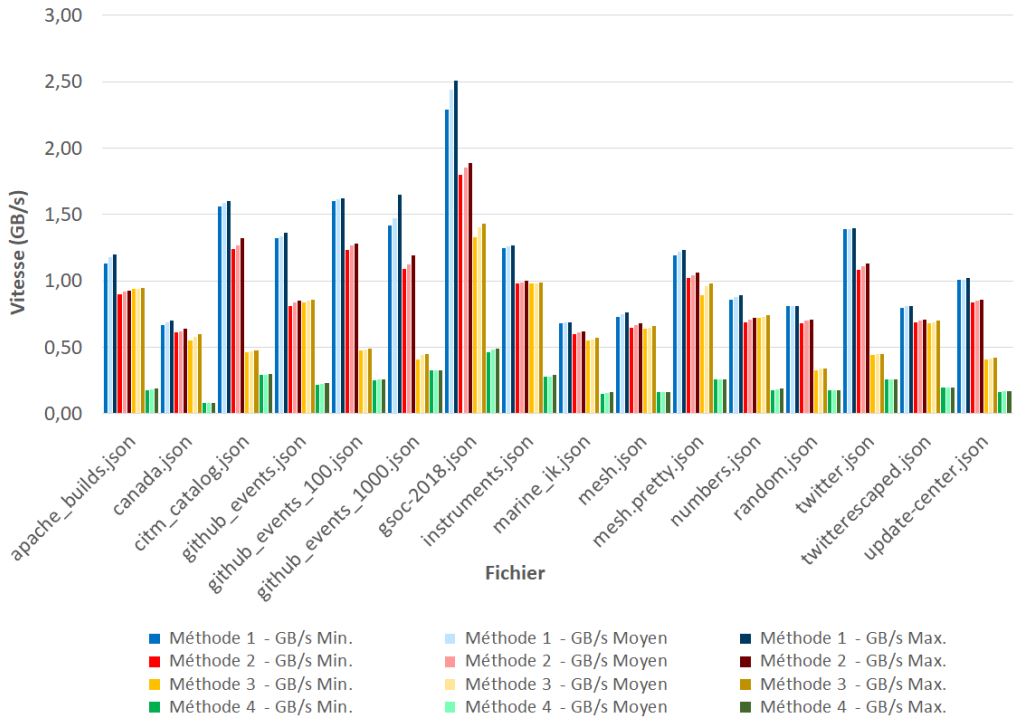

Figure 3.4: Performances minimales et maximales, en GB/s, de l'appel des diverses méthodes d'analyse syntaxique en sous l'architecture Rome

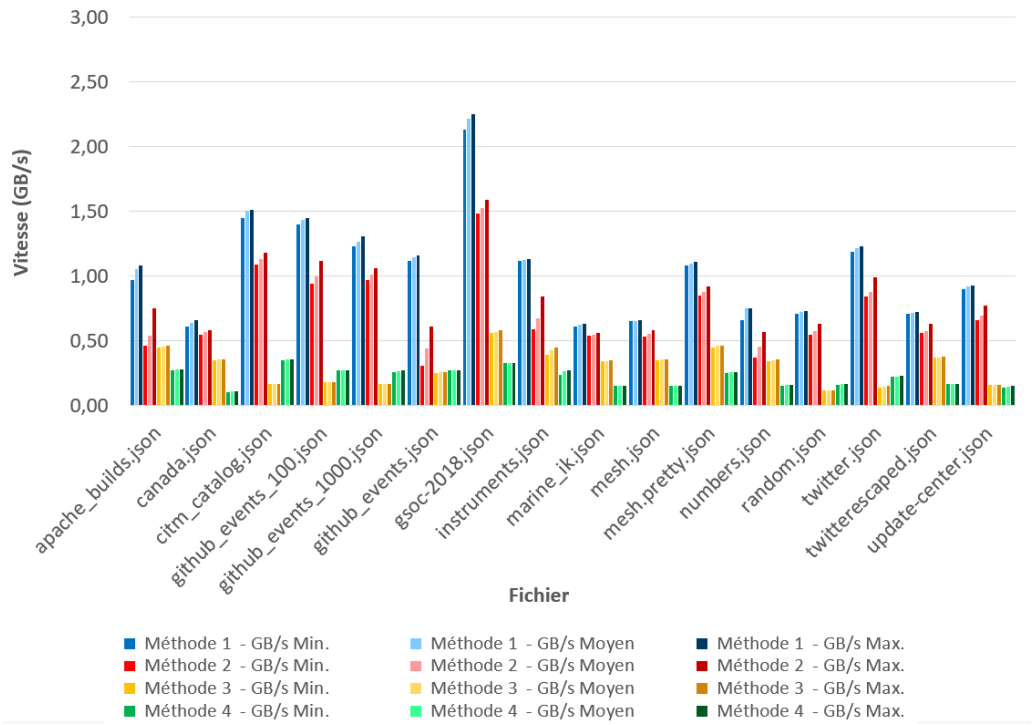

Figure 3.5: Performances minimales et maximales, en GB/s, de l'appel des diverses méthodes d'analyse syntaxique en sous l'architecture Skylake-X

Afin d'offrir une analyse plus juste des performances, les résultats moyens seront comparés entre les diverses méthodes testées. Les tableaux 3.4 et 3.5 présentent les performances moyennes obtenues sous l'architecture Rome par les diverses méthodes d'analyse syntaxique testées, sous forme de tableau. Les résultats présentés par ces tableaux servent de cadre de référence pour les analyses subséquentes des résultats de l'expérimentation.

Fichier	Méthode 1	Méthode 2	Méthode 3	Méthode 4
apache_builds.json	**0,103**	0,132	*0,123*	0,657
canada.json	**3,119**	*3,478*	3,715	26,212
citm_catalog.json	**1,037**	*1,302*	3,518	5,656
github_events.json	**0,046**	0,074	*0,073*	0,277
github_events_100.json	**0,397**	*0,505*	1,325	2,486
github_events_1000.json	**4,350**	*5,688*	14,638	19,302
gsoc-2018.json	**1,300**	*1,714*	2,263	6,547
instruments.json	**0,167**	*0,212*	0,214	0,740
marine_ik.json	**4,151**	*4,617*	5,066	18,197
mesh.json	**0,921**	*1,036*	1,060	4,329
mesh.pretty.json	**1,234**	*1,447*	1,572	5,770
numbers.json	**0,163**	0,203	*0,196*	0,792
random.json	**0,600**	*0,694*	1,444	2,672
twitter.json	**0,433**	*0,541*	1,344	2,332
twitterescaped.json	**0,665**	*0,764*	0,775	2,734
update-center.json	**0,502**	*0,598*	1,226	3,057

Tableau 3.4: Performances, en millisecondes (ms), de l'appel des diverses méthodes d'analyse syntaxique

Fichier	Méthode 1	Méthode 2	Méthode 3	Méthode 4
apache_builds.json	1,18	0,92	0,94	0,18
canada.json	0,69	0,62	0,58	0,08
citm_catalog.json	1,59	*1,27*	0,47	0,29
github_events.json	1,34	0,84	0,85	0,22
github_events_100.json	*1,62*	*1,27*	0,48	0,26
github_events_1000.json	1,47	1,13	0,44	*0,33*
gsoc-2018.json	**2,44**	**1,85**	**1,40**	**0,49**
instruments.json	1,26	0,99	*0,98*	0,28
marine_ik.json	0,69	0,62	0,56	0,16
mesh.json	0,75	0,67	0,65	0,16
mesh.pretty.json	1,22	1,04	0,96	0,26
numbers.json	0,88	0,71	0,73	0,18
random.json	0,81	0,70	0,34	0,18
twitter.json	1,39	1,11	0,45	0,26
twitterescaped.json	0,81	0,70	0,69	0,20
update-center.json	1,01	0,85	0,41	0,17

Tableau 3.5: Performances, en gigaoctet par seconde (GB/s), de l'appel des diverses méthodes d'analyse syntaxique

Pour bien comparer les performances entre les diverses méthodes testées, la variation des performances obtenues en gigaoctet par seconde (GB/s) a été calculée entre diverses méthodes. Les prochains paragraphes présentent les constats observés en comparant les diverses méthodes évaluées.

Tout d'abord, la comparaison des méthodes 1 et 3 permet de déterminer le coût transfert et de conversion d'une chaîne de caractères JavaScript. Pour la méthode 1, le document au format JSON est déjà chargé en mémoire et prêt pour l'analyse, alors que pour la méthode 2 le document est déjà chargé en JavaScript sous forme de chaîne de caractères, puis passé en paramètre à la fonction d'analyse syntaxique. Les résultats présentés au tableau 3.6 démontrent des performances variant de 17% à 239% plus performantes lorsque l'objet est déjà chargé. Ce constat doit être pris en compte dans le développement d'un module de bas niveau utilisant l'interface applicative officielle de la plateforme logicielle Node.js (N-API).

La comparaison de la méthode 2 à la méthode 3 permet d'évaluer s'il est plus performant de charger le document en C++ ou de le passer en paramètre sous forme de chaîne de caractères JavaScript. Les résultats présentés au tableau 3.7 permettent d'affirmer qu'il est en général plus performant de charger un document dans le module C++ avant de procéder à l'analyse syntaxique que de le charger en JavaScript et de le passer par la suite en paramètre pour en analyser syntaxiquement le contenu. Les performances sont nettement supérieures (plus de 100%) pour 6 fichiers et légèrement supérieures (entre 1% et 35%) pour 7 des 16 fichiers.

Il est intéressant de constater que dans certains cas, il est plus performant d'utiliser la troisième méthode. En effet, pour certains fichiers de tests, le passage du document en paramètre est plus rapide que le chargement du fichier. Cela est notamment le cas pour les petits fichiers github_events.json, apache_builds.json et numbers.json. Il est cependant important de soulever que la troisième méthode

Fichier	Méthode 1	Méthode 3		Écart
apache_builds.json	1,18	0,94	+	25%
canada.json	0,69	0,58	+	19%
citm_catalog.json	1,59	0,47	+	**239%**
github_events.json	1,34	0,85	+	58%
github_events_100.json	1,62	0,48	+	*235%*
github_events_1000.json	1,47	0,44	+	*235%*
gsoc-2018.json	2,44	1,40	+	74%
instruments.json	1,26	0,98	+	29%
marine_ik.json	0,69	0,56	+	23%
mesh.json	0,75	0,65	+	15%
mesh.pretty.json	1,22	0,96	+	27%
numbers.json	0,88	0,73	+	20%
random.json	0,81	0,34	+	138%
twitter.json	1,39	0,45	+	210%
twitterescaped.json	0,81	0,69	+	17%
update-center.json	1,01	0,41	+	144%

Tableau 3.6: Variation des performances entre le traitement d'un objet préalablement chargé en C++ et en JavaScript.

n'évalue pas le chargement d'un fichier, mais seulement le passage de son contenu. Il est donc possible d'affirmer que lorsque le document à analyser syntaxiquement est relativement petit (moins de 200 kilooctets) et qu'il est possible de l'analyser sans l'utilisation d'un fichier sur le disque, il sera plus avantageux d'utiliser la troisième méthode.

Fichier	Méthode 2	Méthode 3		Écart
apache_builds.json	0,92	0,94	-	*3%*
canada.json	0,62	0,58	+	7%
citm_catalog.json	1,27	0,47	+	170%
github_events.json	0,84	0,85	-	1%
github_events_100.json	1,27	0,48	+	163%
github_events_1000.json	1,13	0,44	+	156%
gsoc-2018.json	1,85	1,40	+	32%
instruments.json	0,99	0,98	+	1%
marine_ik.json	0,62	0,56	+	10%
mesh.json	0,67	0,65	+	3%
mesh.pretty.json	1,22	1,04	+	9%
numbers.json	0,71	0,73	-	**4%**
random.json	0,70	0,34	+	106%
twitter.json	1,11	0,45	+	148%
twitterescaped.json	0,70	0,69	+	2%
update-center.json	0,85	0,41	+	106%

Tableau 3.7: Variation des performances entre le chargement de fichier en C++ et le passage de l'objet en JavaScript

Afin de comparer résultats obtenus aux résultats attendus tels que définis dans le cadre méthodologique présenté au chapitre 2, les performances des deux méthodes d'analyse syntaxique les plus susceptibles d'être concrètement utilisées, soit l'analyse syntaxique à partir d'un chemin d'accès à un fichier au format JSON (méthode 2) et l'analyse syntaxique d'une chaîne de texte représentant un document JSON valide (méthode 3), ont été comparées à celles de la fonction native `JSON.parse()`. Les résultats sont présentés au tableau 3.8.

Fichier	Méthode 4 JSON.parse()	Méthode 2		Écart	Méthode 3		Écart
apache_builds.json	0,18	0,92	+	396%	0,94	+	*411%*
canada.json	0,08	0,62	+	**673%**	0,58	+	**623%**
citm_catalog.json	0,29	1,27	+	336%	0,47	+	61%
github_events.json	0,22	0,84	+	274%	0,85	+	279%
github_events_100.json	0,26	1,27	+	391%	0,48	+	87%
github_events_1000.json	0,33	1,13	+	241%	0,44	+	33%
gsoc-2018.json	0,49	1,85	+	280%	1,40	+	188%
instruments.json	0,28	0,99	+	252%	0,98	+	248%
marine_ik.json	0,16	0,62	+	286%	0,56	+	251%
mesh.json	0,16	0,67	+	317%	0,65	+	306%
mesh.pretty.json	0,26	1,04	+	300%	0,96	+	268%
numbers.json	0,18	0,71	+	290%	0,73	+	304%
random.json	0,18	0,70	+	289%	0,34	+	89%
twitter.json	0,26	1,11	+	328%	0,45	+	73%
twitterescaped.json	0,20	0,70	+	250%	0,69	+	245%
update-center.json	0,17	0,85	+	*401%*	0,41	+	143%

Tableau 3.8: Variation des performances entre la fonction native `JSON.parse()` et les deux méthodes le plus susceptibles d'être utilisées

L'analyse syntaxique à partir d'un chemin d'accès menant à un fichier de format JSON valide offre une augmentation des performances supérieure à 200% pour l'ensemble des fichiers testés en comparaison à la fonction native `JSON.parse()`, la variation des performances se situant entre 250% et 673% , ce qui représente d'excellents résultats. De plus, la méthode native `JSON.parse()` ne prend que du texte en paramètre, ce qui signifie que pour analyser syntaxiquement un fichier, il est nécessaire le charger d'abord. Le temps de chargement du document en JavaScript n'ayant pas été évalué dans les performances présentées, il est possible de supposer que l'écart est encore plus élevé lorsque l'utilisateur doit analyser syntaxiquement un fichier sur le disque. L'analyse syntaxique à partir d'une chaîne de texte contenant un document de format JSON valide présente des performances inférieures au chargement par fichier, mais offre tout de même d'excellents résultats (plus de 75% d'augmentation) pour 11 fichiers, de bons résultats (entre 50% et 75% d'augmentation) pour 4 fichiers et des résultats moyens pour l'un des fichiers testés. Dans l'ensemble, il est possible de conclure que ces résultats, variant de 33% à 623%, sont excellents.

Toujours dans le but de valider l'atteinte des résultats attendus, les performances de la méthode 1 ont été comparées aux performances de la bibliothèque simdjson. Dans ce cas, seule la variation entre la méthode 1 et la bibliothèque simdjson a été analysée puisqu'elle représente la seule méthode effectuant un calcul des performances comparable aux bancs d'essai des méthodes d'analyse syntaxique implémentées par le module simdjsonJS. Le tableau 3.9 présente les écarts de variation obtenus.

Cette analyse permet d'évaluer le prix en performances de l'intégration de la bibliothèque `simdjson` dans un objet encapsulé par un module `simdjsonJS` tel que définit par l'interface officielle fournie à cet effet, N-API. La variation représente une diminution inférieure à 10% pour 3 fichiers ainsi qu'une diminution entre 10%

Fichier	simdjson.parse()	Méthode 1		Écart
apache_builds.json	1,38	1,18	-	14%
canada.json	0,77	0,69	-	10%
citm_catalog.json	1,81	1,59	-	12%
github_events.json	1,70	1,34	-	21%
github_events_100.json	1,85	1,62	-	12%
github_events_1000.json	1,83	1,47	-	20%
gsoc-2018.json	2,73	2,44	-	11%
instruments.json	1,54	1,26	-	18%
marine_ik.json	0,76	0,69	-	*9%*
mesh.json	0,83	0,75	-	10%
mesh.pretty.json	1.32	1,22	-	**8%**
numbers.json	0.98	0,88	-	10%
random.json	0.92	0,81	-	12%
twitter.json	1,62	1,39	-	14%
twitterescaped.json	0,90	0,81	-	10%
update-center.json	1,13	1,01	-	*9%*

Tableau 3.9: Variation des performances entre la bibliothèque simdjson et la méthode 1

et 25% pour les 13 autres fichiers testés. Selon les résultats attendus fixés dans le cadre méthodologique au chapitre 2, l'expérimentation présente de bons résultats.

Il est également possible d'observer que les performances obtenues par la méthode 3 sont supérieures à celles obtenues par la fonction `simdjson.lazyParse1Depth()` évaluées dans la première expérimentation, et ce, pour la majorité des fichiers de tests. Afin d'assurer une comparaison juste des performances, les méthodes

`lazyParse()` et `lazyPase1Depth()` ont été mises à jour avec la version 0.3.1 de la bibliothèque simdjson puis intégrées en tant que fonctions statiques dans l'objet simdjsonJS. Le graphique 3.6 présente les résultats des fonctions intégrées en comparaison avec la méthode 3 puisque celle-ci représente la seule méthode prenant offrant la possibilité de fournir les mêmes paramètres d'entrées.

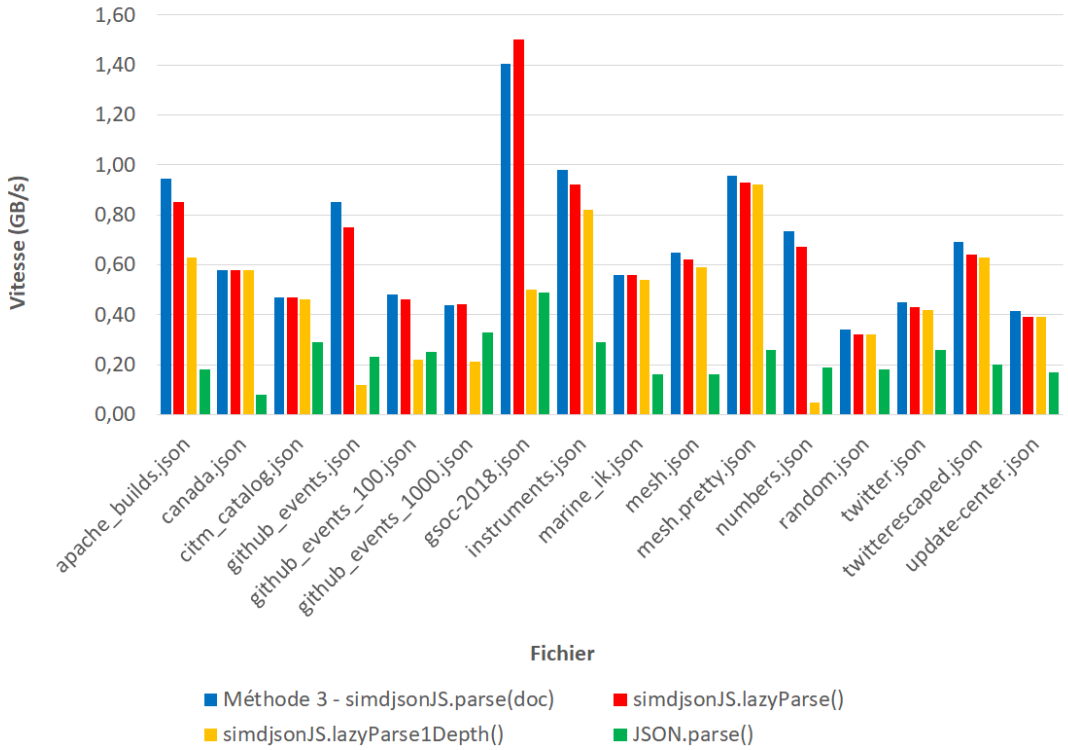

Figure 3.6: Comparaison des performances de la méthode 3 aux fonctions statiques `simdjsonJS.lazyParse()`, `simdjsonJS.lazyPase1Depth()` et `JSON.parse()`

Dans l'ensemble, il est possible de conclure que la méthodologie d'encapsulation de l'objet est celle qui, en utilisant les modules officiels proposés par Node.js, offre les meilleurs résultats en performances. Bien que cette solution diminue les performances de la bibliothèque simdjson (diminution évaluée entre 8% et 21% sur les fichiers de tests sélectionnés), elle offre cependant des performances supérieures à la fonction native JSON.parse(), ce qui en fait une solution intéressante à utiliser et exploiter.

Il serait peut-être possible d'obtenir de meilleures performances en contournant le module officiel N-API et en exploitant directement les divers moteurs JavaScript existants tels que V8, SpiderMonkey ou Chakra. Cette pratique est cependant non recommandée par la plateforme logicielle Node.js puisque l'interface N-API est à ce jour la seule à garantir une stabilité de niveau de l'interface applicative binaire (ABI), ce qui permet une compatibilité du module parmi tous les moteurs JavaScript compatibles avec la plateforme logicielle Node.js, et ce, en considérant également les mises à jour majeures à venir (Node.js, 2021a).

Maintenant que les performances de l'analyse syntaxique démontrent que les performances sont bonnes, il est intéressant de poursuivre l'évaluation des résultats des méthodes implémentées afin de valider si une utilisation courante de l'objet demeure possible et performante. Les prochaines sections présentent les résultats obtenus par les fonctionnalités implémentés par la solution proposée dans le cadre de cette expérimentation.

3.2.3.2 Affichage de l'objet

Les objets analysés syntaxiquement puis affichés dans la console à l'aide de la commande console.log() sont comparables à ceux obtenus par la fonction native JSON.parse(). Le contenu de l'objet simdjsonJS du fichier de données

themoviedb.json a été comparé au contenu de l'objet JSON afin d'évaluer le taux de différences entre les affichages de ces deux objets. La figure 3.7 présente un échantillon des deux premiers objets provenant du fichier de données themoviedb.json.

Figure 3.7: Échantillon des objets analysés syntaxiquement sous simdjson (gauche) et JSON (droite)

Les résultats complets obtenus par la commande console.log() sur les objets évalués ont été comparés à l'aide de la commande `diff` afin d'identifier toutes les différences pour s'assurer qu'aucune d'elles ne passe inaperçue. La commande `diff` a identifié 4 différences dans son rapport présenté par la figure 3.8.

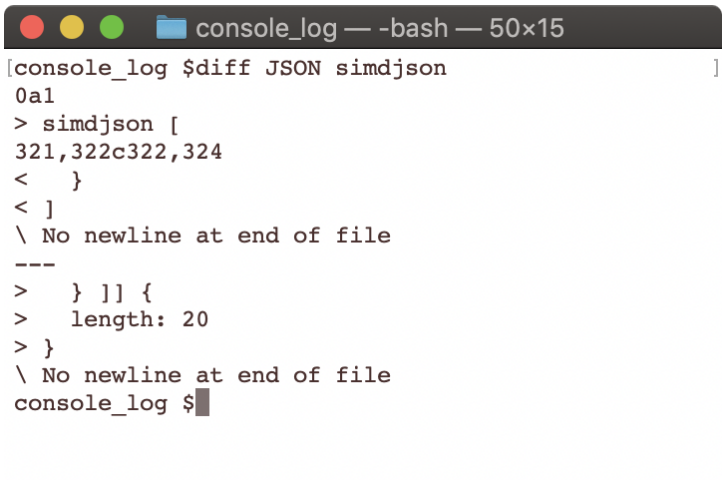

Figure 3.8: Rapport de différences obtenu à l'aide de la commande `diff`

Ces différences sont associées à une particularité de conception dans la solution mise en place dans cette recherche. Elles permettent en effet de constater que l'objet traité a été encapsulé. De plus, puisque la taille est calculée au même moment que l'analyse syntaxique, il est également possible d'accéder directement à la valeur de cette propriété.

Les différences identifiées n'ont pas d'incidence sur la structure et le contenu du document. On peut donc conclure que ces différences sont négligeables et que les deux méthodes offrent des résultats comparables. Il est ainsi possible pour un utilisateur de bien comprendre la structure du document analysé lorsqu'il en affiche le contenu.

3.2.3.3 Taille

Lorsque l'objet analysé est un tableau, il est intéressant d'offrir à l'utilisateur la possibilité d'obtenir la taille de celui-ci. Il est aussi important de s'assurer que l'interface développée offre des fonctionnalités similaires à l'objet JSON natif. Le

tableau 3.10 présente les résultats obtenus par la propriété `length` implémentée comparés à la propriété native `length` du protocole Array (MDN Web Docs, 2019b).

Fichier	simdjsonJS.length	JSON.length
apache_builds.json	undefined	undefined
canada.json	undefined	undefined
citm_catalog.json	undefined	undefined
github_events.json	30	30
github_events_100.json	100	100
github_events_1000.json	1000	1000
gsoc-2018.json	undefined	undefined
instruments.json	undefined	undefined
marine_ik.json	undefined	undefined
mesh.json	undefined	undefined
mesh.pretty.json	undefined	undefined
numbers.json	10001	10001
random.json	undefined	undefined
twitter.json	undefined	undefined
twitterescaped.json	undefined	undefined
update-center.json	undefined	undefined

Tableau 3.10: Représentation des résultats lors de l'appel de la propriété `length`

Dans les deux cas, les résultats obtenus sont identiques, ce qui correspond totalement à l'objectif du point de vue de la convivialité de la fonctionnalité implémentée.

Le tableau 3.11 présente les performances obtenues en millisecondes lors de l'appel de la fonction `length`, sous forme de tableau. Dans le cas des documents qui

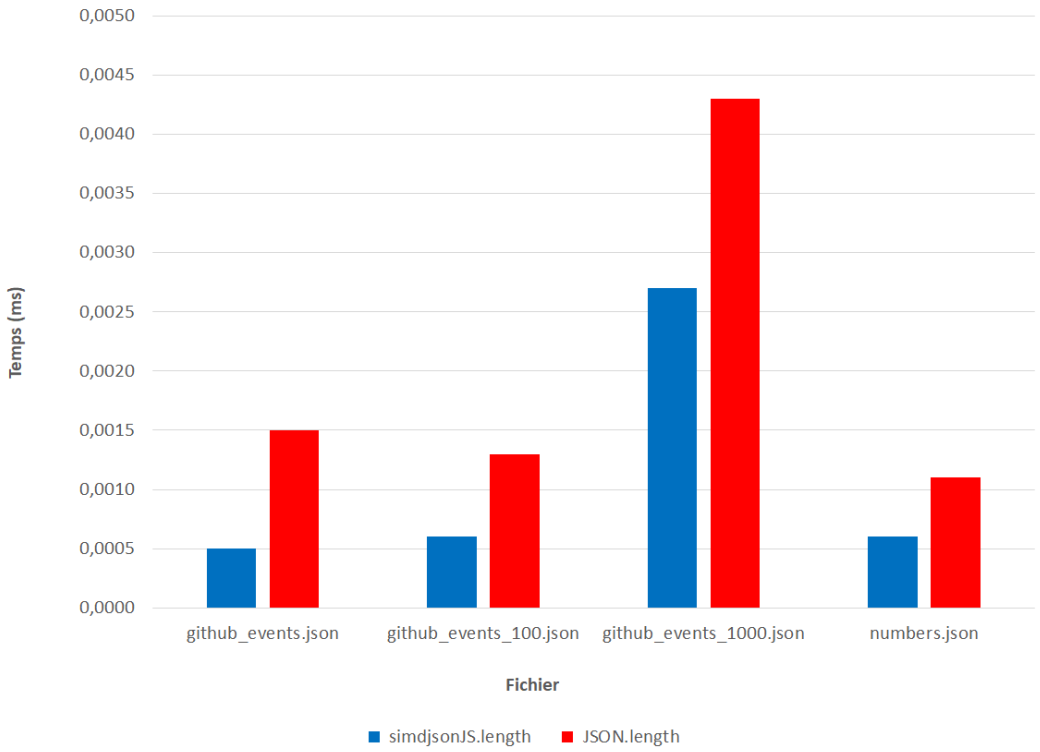

Figure 3.9: Performances, en millisecondes, de l'appel de la propriété `length` sur les documents structurés sous forme de tableaux

ne sont pas structurés en tableaux, les performances attendues devraient être similaires puisque l'objet n'est pas parcouru. Dans le cas des tableaux, l'objectif est d'offrir de meilleures performances que la propriété native `length` du prorotype Array. Puisque les résultats attendus diffèrent en fonction du type d'élément analysé syntaxiquement, les résultats sont donc présentés dans deux graphiques. La figure 3.9 présente le graphique des performances obtenues en millisecondes lors de l'appel de la fonction `length` sur les documents représentant une structure organisée en tableau alors que la figure 3.10 présente les performances sur les autres types d'objets.

Les performances obtenues pour l'obtention de la taille de documents structurées en tableau sont excellentes. Dans les quatre cas où les documents représentent des

Fichier	simdjsonJS.length	JSON.length
apache_builds.json	0,0014	**0,0004**
canada.json	0,0018	*0,0007*
citm_catalog.json	0,0007	0,0015
github_events.json	**0,0005**	0,0015
github_events_100.json	*0,0006*	0,0013
github_events_1000.json	0,0027	0,0043
gsoc-2018.json	0,0008	0,0015
instruments.json	0,0007	0,0013
marine_ik.json	0,0012	0,0015
mesh.json	0,0007	0,0012
mesh.pretty.json	0,0007	0,0013
numbers.json	0,0006	0,0011
random.json	0,0007	0,0012
twitter.json	0,0007	0,0025
twitterescaped.json	0,0007	0,0025
update-center.json	0,0007	0,0026

Tableau 3.11: Performances, en millisecondes, de l'appel de la propriété `length`

tableaux, les performances obtenues par la fonctionnalité implémentée sont plus rapides que la fonctionnalité native équivalente.

Dans le cas des fichiers ne représentant pas une structure en tableau, les performances sont moyennes. Les performances sont moins bonnes dans les premiers fichiers testés et meilleures dans les derniers. Afin de tenter d'expliquer la variation des temps obtenus, l'ordre des fichiers a été inversé dans le script de banc d'essai afin de valider si les résultats étaient constants pour un même fichier donné. Les

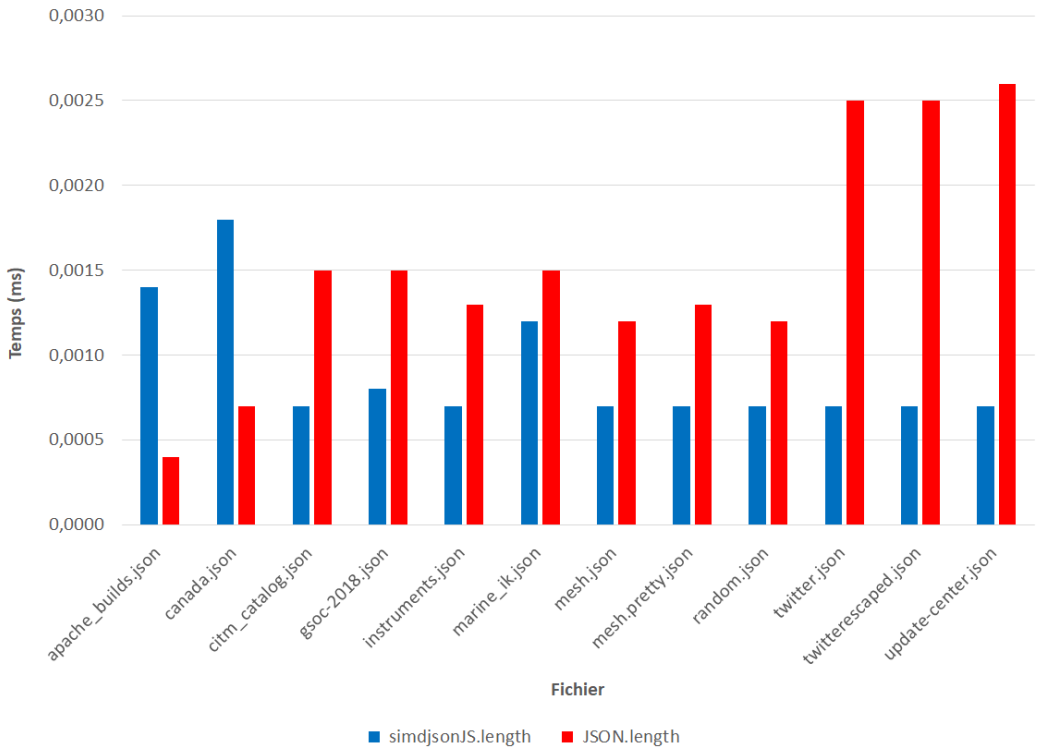

Figure 3.10: Performances, en millisecondes, de l'appel de la propriété `length` sur les documents structurés dans un format autre qu'en tableau

résultats présentés à la figure 3.11 permettent d'observer que dans les deux cas, les performances varient. Dans le cas des objets simdjsonJS, le temps d'obtention semble plus élevé lors des premiers appels, alors que dans le cas des objets JSON, les temps se dégradent après une certaine utilisation.

Considérant qu'il est peu probable que l'utilisateur utilise la propriété `length` sur les documents qui ne sont pas structurés en tableau, et que ceux-ci démontrent des résultats variables sous les deux méthodes évaluées nous pouvons conclure qu'il demeure plus intéressant de calculer la taille du document en utilisation la fonctionnalité implémentée par l'objet simdjsonJS, surtout si l'on considère que l'utilisateur doit d'abord analyser syntaxiquement le document et que les méthodes d'analyses testées ont démontré de meilleures performances que la méthode JSON.

parse() native.

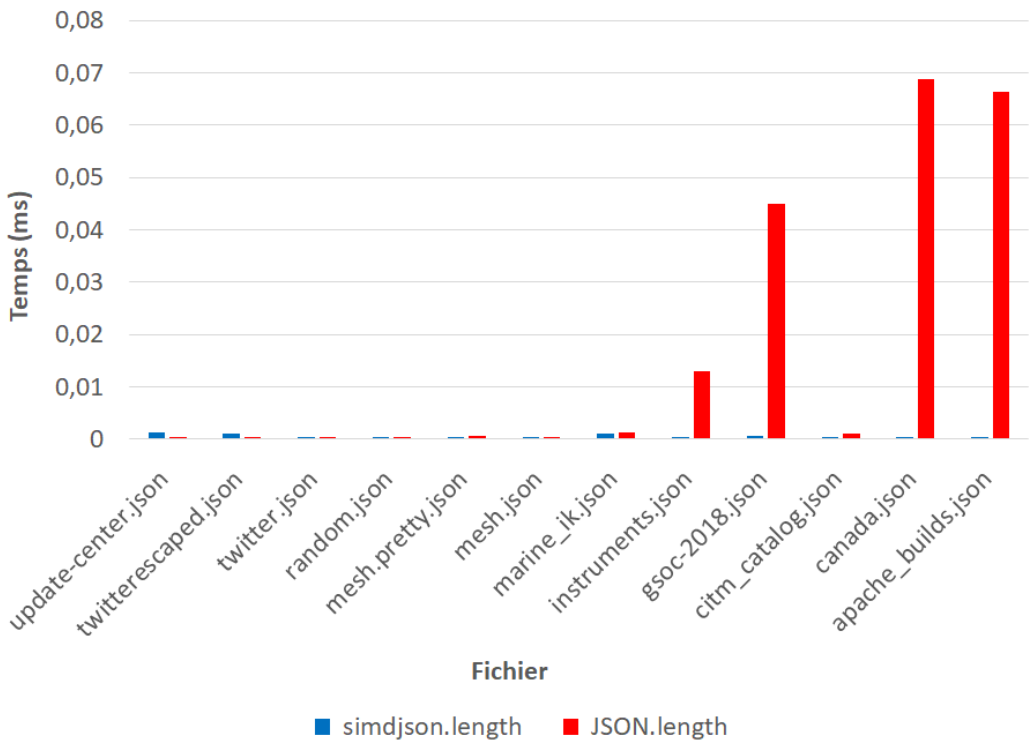

Figure 3.11: Performances, en millisecondes, de l'appel de la propriété `length` sur les documents structurés dans un format autre qu'en tableau, en ordre inverse

3.2.3.4 Liste des propriétés

Selon la documentation officielle de JavaScript, la fonction native Object.keys() renvoie un tableau contenant les noms des propriétés propres à un objet (qui ne sont pas héritées via la chaîne de prototypes) et qui sont énumérables (MDN Web Docs, 2019e). Lorsque le document analysé représente un objet, les propriétés représentent alors la liste des attributs de cet objet. Lorsque le document représente un tableau, les propriétés correspondent aux indices de ce dernier.

Le tableau 3.12 présente le nombre de propriétés obtenues en comparaison à la fonction native Object.keys() que l'on peut appliquer sur un objet de type JSON. Dans les deux cas, les résultats obtenus sont identiques, ce qui correspond totalement à l'objectif du point de vue de la convivialité de la fonctionnalité implémentée. Le détail des propriétés générées peut être consulté à l'Annexe B.

Fichier	simdjsonJS.keys()	Object.keys(JSON)
apache_builds.json	15	15
canada.json	2	2
citm_catalog.json	11	11
github_events.json	30	30
github_events_100.json	100	100
github_events_1000.json	1000	1000
gsoc-2018.json	1264	1264
instruments.json	9	9
marine_ik.json	7	7
mesh.json	8	8
mesh.pretty.json	8	8
numbers.json	10001	10001
random.json	4	4
twitter.json	2	2
twitterescaped.json	2	2
update-center.json	6	6

Tableau 3.12: Représentation des résultats de l'appel de la méthode `keys()`

Afin d'optimiser les performances de la fonctionnalité implémentée et d'éviter de devoir régénérer des propriétés déjà obtenues lors d'un premier appel, le résultat obtenu par l'appel de la méthode `keys()` est conservé en mémoire dans le module développé en C++. Deux méthodes ont été utilisées afin de mesurer les performances de la méthode `keys()` pour permettre d'évaluer le temps de construction et le temps moyen des appels subséquents.

- Méthode 1 : création d'un nouvel objet analysé et génération des propriétés pour chacune des itérations. Cette méthode vise à calculer le temps moyen nécessaire pour la première génération des propriétés d'un objet.
- Méthode 2 : création d'un nouvel objet et itérations successives pour l'obtention des propriétés. Cette méthode vise quant à elle à calculer le temps moyen nécessaire pour effectuer plusieurs appels permettant d'obtenir les propriétés sur un même objet préalablement analysé.

Le tableau 3.13 présente les performances obtenues lors de l'appel de la fonction permettant de générer les propriétés selon la méthode employée, sous forme de tableau. Les temps supérieurs à 0,1 milliseconde sont mis en évidence.

En analysant le nombre de propriétés en fonction des performances, il est possible de supposer qu'il y a une certaine corrélation entre le temps requis pour obtenir les propriétés et le nombre de propriétés retournées par la fonctionnalité implémentée. En effet, plus il y a de propriétés, plus le temps semble être significatif. La figure 3.12 présente un graphique servant d'indice de mesure pour cette corrélation.

Le graphique permet en effet d'observer une corrélation linéaire entre le temps et le nombre de propriétés dans l'application de la méthode 1 pour la fonctionnalité implémentée ainsi que son équivalent natif. Cette corrélation représente une complexité algorithmique d'ordre O(n) dans ces deux cas. Cela permet d'établir une comparabilité entre les deux, avec des performances légèrement meilleures pour la fonction native.

	Méthode 1		Méthode 2	
Fichier	simdjsonJS.keys()	Object.keys(JSON)	simdjsonJS.keys()	Object.keys(JSON)
apache_builds.json	0,015	**0,001**	**0,004**	**0,001**
canada.json	0,009	*0,002*	**0,004**	*0,002*
citm_catalog.json	0,013	*0,002*	**0,004**	*0,002*
github_events.json	0,010	0,006	**0,004**	0,007
github_events_100.json	0,022	0,015	**0,004**	0,025
github_events_1000.json	0,195	0,131	**0,004**	0,303
gsoc-2018.json	0,545	0,166	**0,004**	0,335
instruments.json	0,011	*0,002*	**0,004**	0,112
marine_ik.json	0,014	*0,002*	**0,004**	0,007
mesh.json	0,010	0,002	**0,004**	0,003
mesh.pretty.json	0,011	*0,002*	*0,006*	0,003
numbers.json	1,55	1,435	*0,006*	1,528
random.json	0,009	**0,001**	**0,004**	0,005
twitter.json	*0,008*	*0,002*	**0,004**	0,052
twitterescaped.json	**0,007**	*0,002*	**0,004**	0,003
update-center.json	0,010	*0,002*	**0,004**	0,023

Tableau 3.13: Performances, en millisecondes, de l'appel de la méthode `keys()`

L'analyse de la seconde méthode apporte cependant d'autres constats lors de l'évaluation de la corrélation. En effet, la fonction native ne semble pas tirer avantage d'appels subséquents et ses performances semblent toujours dépendre du nombre de propriétés de l'objet. Sa complexité demeure d'ordre O(n). Dans le cas de la fonction implémentée, elle offre quant à elle un résultat dans un temps stable de complexité d'ordre O(1) et donc, de meilleures performances.

Il est donc possible de conclure que les performances obtenues pour la génération des propriétés varient selon son contexte d'utilisation. Il est cependant important de noter qu'il est peu probable, dans une utilisation courante, qu'un utilisateur se serve de la génération des propriétés plus d'une fois. Si nous supposons que l'utilisateur se servira de la fonction une seule fois, les résultats sont peu concluants

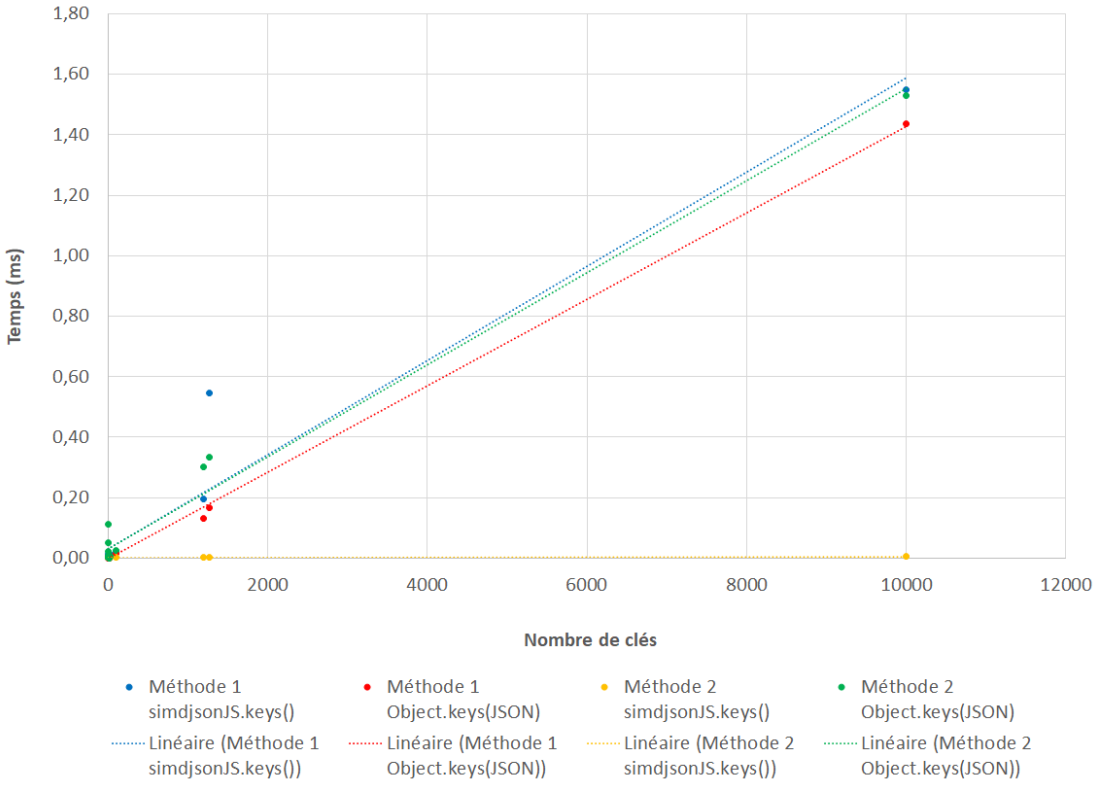

Figure 3.12: Corrélation entre le nombre de propriétés et les performances, en millisecondes (ms)

puisque les temps obtenus sont tous supérieurs à la fonction native `Object.keys()`. Il est cependant important de noter que cette différence se situe entre 0,004 et 0,379 milliseconde, ce qui est peu significatif.

Dans un contexte d'utilisation normale, l'utilisateur devra cependant d'abord analyser syntaxiquement le document avant d'en générer les propriétés. La figure 3.13 présente les résultats des deux méthodes d'analyse syntaxique les plus susceptibles d'être utilisées, soit par passage du chemin d'accès en paramètre (méthode 2) et par passage du document sous forme de chaîne de caractères (méthode 3) combinés aux résultats de la génération des propriétés en comparaison aux fonctionnalités natives équivalentes.

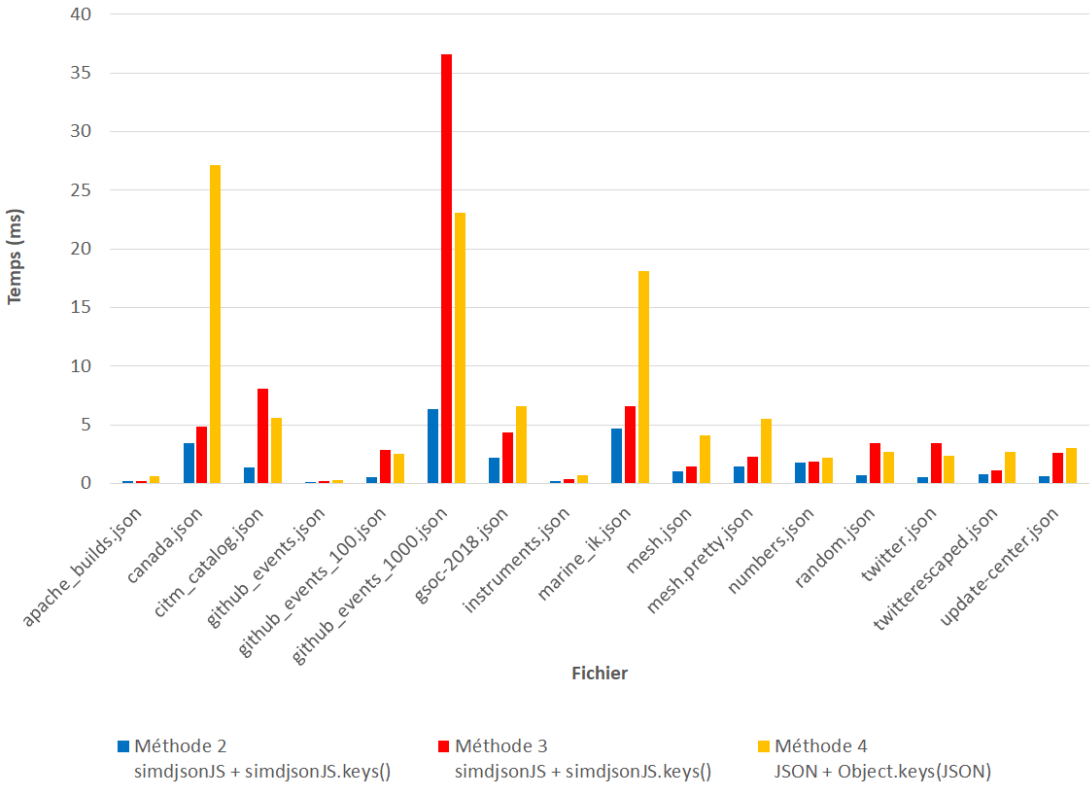

Figure 3.13: Performances, en millisecondes (ms), de l'appel des diverses méthodes d'analyse syntaxique combiné à l'appel de la méthode `keys()`

Le graphique présenté permet de constater que dans l'ensemble des cas, l'utilisation de la méthode 2 semble plus performante que la fonction native. Il ne faut cependant pas oublier que temps de chargement du fichier peut varier et qu'il est difficile à mesurer puisqu'il peut dépendre de nombreux facteurs, tels que le type de disque dur, les stratégies mises en place par le système d'exploitation pour en accélérer le chargement, etc. Cependant, dans une situation où l'utilisateur doit charger un fichier, et que nous estimons que les temps pour le charger en JavaScript ou C++ sont similaires, il est possible d'affirmer que la fonctionnalité implémentée sera plus performante.

La méthode implémentée chargeant d'abord le fichier en JavaScript pour le passer par la suite en paramètre pour une analyse syntaxique propose des performances moins intéressantes. Il demeure plus performant d'utiliser cette méthode que les fonctions natives pour 11 des 16 de tests utilisés. Par contre, les performances sont nettement moins intéressantes pour le fichier de tests github_events_1000.json.

3.2.3.5 Parcours

Selon la documentation officielle de JavaScript, Le protocole "itérable" permet aux objets JavaScript de définir ou de personnaliser leur comportement lors d'une itération, par exemple la façon dont les valeurs seront parcourues avec une boucle for...of. Certains types natifs tels que Array ou Map possèdent un comportement itératif par défaut, d'autres types, comme Object, n'ont pas ce type de comportement (MDN Web Docs, 2019d). Lorsque le document analysé représente un tableau, il est donc souhaitable d'offrir la possibilité à un utilisateur de parcourir le document à l'aide de tels mécanismes afin de permettre une utilisation similaire à un objet de type Array natif. La figure 3.14 présente un échantillon du parcours des objets provenant du fichier de données themoviedb.json.

Le parcours d'objets structurés en tableau présente des résultats équivalents au parcours d'un objet JSON natif. Pour maximiser les performances, la conversion en JavaScript étant nécessaire pour produire le résultat désiré, la technique de chargement paresseux mis en oeuvre dans la première expérimentation a été réutilisée. Seul le premier niveau du tableau est ainsi chargé, les objets de niveaux supérieurs étant associés à une méthode d'obtention qui est appelée lorsque l'utilisateur tente de récupérer la valeur d'un objet imbriqué. Cette différence s'observe par la propriété `genre_ids` de l'échantillon présenté.

Figure 3.14: Échantillon des objets parcourus sous simdjson (gauche) et JSON (droite)

Figure 3.15: Propriétés `genre_ids` des objets parcourus sous simdjson (gauche) et JSON (droite)

Lorsque l'utilisateur tente de récupérer une propriété d'un niveau non chargé, la plateforme logicielle appelle la méthode d'obtention fournie par le module intermédiaire implémenté et récupère le contenu demandé en tirant profit du pointeur JSON. Le tableau 3.14 présente les résultats obtenus lors de l'affichage de la propriété `genre_ids` des objets présents dans le fichier de données themoviedb.json. Ce tableau permet de constater que les résultats sont identiques. L'exemple de manipulation d'un objet à l'Annexe A permet de comprendre comment il est possible d'obtenir un objet imbriqué dans une boucle de type for...of.

Afin d'évaluer les performances du parcours implémenté dans le cadre de cette expérimentation, deux méthodes ont été utilisées. La première méthode vise à calculer le temps moyen nécessaire pour le premier parcours des éléments, alors que la seconde méthode vise à calculer le temps moyen nécessaire pour effectuer plusieurs parcours sur un objet de type tableau préalablement analysé. Le tableau 3.15 présente les performances obtenues lors de l'utilisation de la bouche for...of permettant de parcourir les éléments d'un document représentant un tableau. Les temps supérieurs à 0,5 milliseconde sont mis en évidence.

Afin de mieux comparer les performances, celles-ci sont également présentées sous forme graphique. Dans le cas des documents qui ne sont pas structurés en tableaux, les performances attendues devraient être similaires puisque l'objet n'est pas parcouru. Dans le cas des tableaux, l'objectif est d'offrir de meilleures performances que le parcours d'un objet JSON natif. Puisque les résultats attendus diffèrent en fonction du type d'élément analysé syntaxiquement, les résultats sont donc présentés dans deux graphiques. La figure 3.16 présente le graphique des performances obtenues en millisecondes lors du parcours de documents structurés en tableaux alors que la figure 3.17 présente les performances sur les autres types d'objets.

Indice	simdjsonJS.item.genre_ids	JSON.item.genre_ids
1	[18, 878]	[18, 878]
2	[12, 16, 35, 9648, 10751]	[12, 16, 35, 9648, 10751]
3	[28, 18, 878]	[28, 18, 878]
4	[35, 10749]	[35, 10749]
5	[28, 35, 878, 10751]	[28, 35, 878, 10751]
6	[28, 18, 53]	[28, 18, 53]
7	[28, 35, 80]	28, 35, 80]
8	[28, 18, 36, 53, 10752]	[28, 18, 36, 53, 10752]
9	[28, 12, 16, 14, 878]	[28, 12, 16, 14, 878]
10	[12, 16, 10751]	[12, 16, 10751]
11	[12, 16, 35, 14, 10751]	[12, 16, 35, 14, 10751]
12	[28, 12, 878]	[28, 12, 878]
13	[37]	[37]
14	[12, 18, 10751]	[12, 18, 10751]
15	[28, 12, 878]	[28, 12, 878]
16	[28, 80, 53]	[28, 80, 53]
17	[12, 16, 35, 10751]	[12, 16, 35, 10751]
18	[35, 18, 53]	[35, 18, 53]
19	[53, 878, 27]	[53, 878, 27]
20	[80, 18, 53]	[80, 18, 53]

Tableau 3.14: Valeurs associées à la propriété `genres_id` de l'item itéré

Les performances obtenues pour les fichiers représentant des objets qui ne sont pas structurés en tableau sont excellentes. Peu importe le document analysé, les résultats de la fonctionnalité implémentée sont inférieurs à la fonctionnalité native.

	Méthode 1		Méthode 2	
Fichier	simdjson for...of	JSON for...of	simdjson for...of	JSON for...of
apache_builds.json	**0,035**	0,111	*0,022*	*0,061*
canada.json	0,073	0,132	*0,022*	0,068
citm_catalog.json	0,071	0,116	*0,022*	0,117
github_events.json	0,546	*0,058*	**0,020**	**0,059**
github_events_100.json	1,807	0,063	0,043	0,082
github_events_1000.json	16,985	0,110	0,043	0,083
gsoc-2018.json	0,076	0,120	0,024	0,099
instruments.json	*0,048*	0,110	0,024	0,155
marine_ik.json	0,087	0,146	0,024	0,127
mesh.json	0,075	**0,014**	0,030	0,100
mesh.pretty.json	0,076	0,115	0,030	0,077
numbers.json	9,230	0,671	0,193	0,574
random.json	0,075	0,115	0,030	0,122
twitter.json	0,074	0,119	0,024	0,147
twitterescaped.json	0,076	0,117	0,024	0,121
update-center.json	0,075	0,117	0,030	*0,061*

Tableau 3.15: Performances, en millisecondes, du parcours de l'objet

Les performances obtenues pour les fichiers représentant des documents structurés en tableau sont non-concluantes. Dans un contexte d'utilisation normale, la boucle for...of sera principalement utilisée sur ce type d'objet et les temps nécessaires pour effectuer ce traitement sont significativement supérieurs à l'itération d'un objet JSON natif.

Pour tenter de diminuer cette problématique, le résultat est cependant conservé pour une utilisation future et dans ces cas précis, l'utilisation de la boucle for...of est plus légèrement plus rapide que son équivalent natif. Malgré le fait qu'il soit

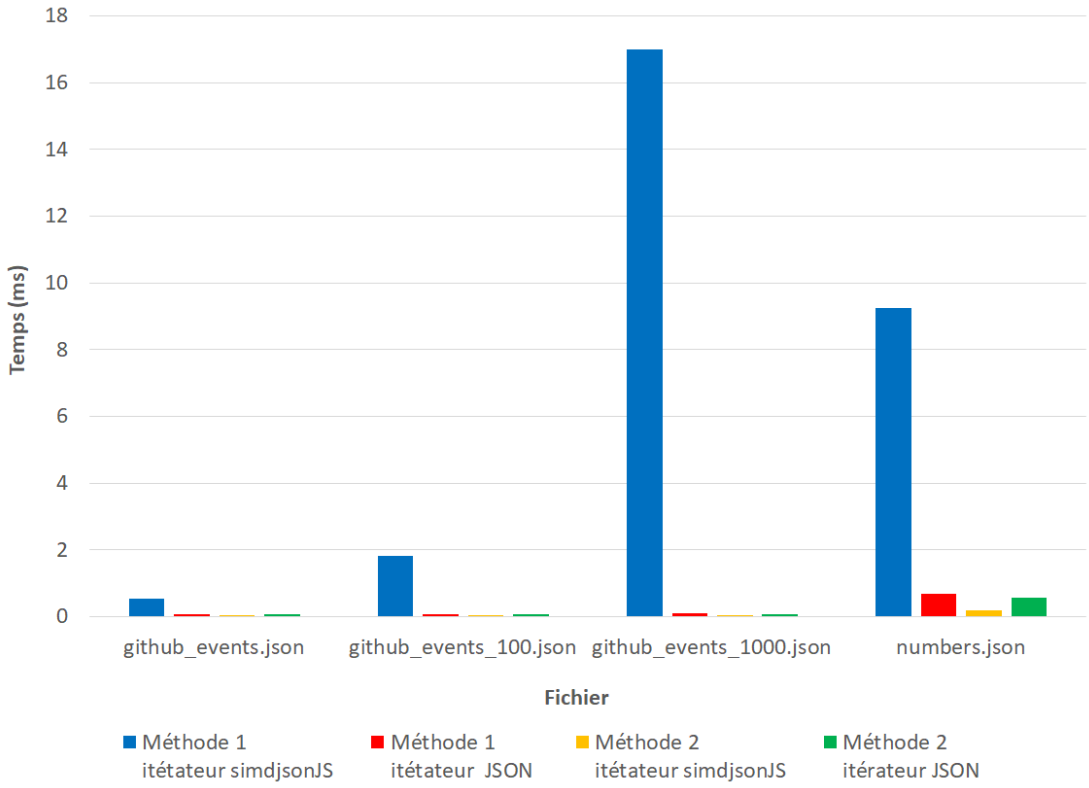

Figure 3.16: Performances, en millisecondes, du parcours de tableaux

peu probable que l'utilisateur appelle ce type de boucle plus d'une fois pour un même objet, les résultats ont tout de même été analysés afin de vérifier que les pertes de performances de l'itération ne demeurent pas pour les appels subséquents. Les résultats présentés permettent d'affirmer que les résultats subséquents sont plus performants par la fonctionnalité implémentée que par son équivalent natif.

Afin d'expliquer les résultats les plus élevés qui surviennent lors de la génération de l'itérateur sur les tableaux, les temps en millisecondes des 100 premiers parcours ont été récupérés, et ce, pour les deux méthodes d'analyse identifiées. Les figures 3.18 et 3.19 présentent le temps d'exécution des 100 premières itérations de la première méthode. Elles permettent de présenter le temps requis pour parcourir un document

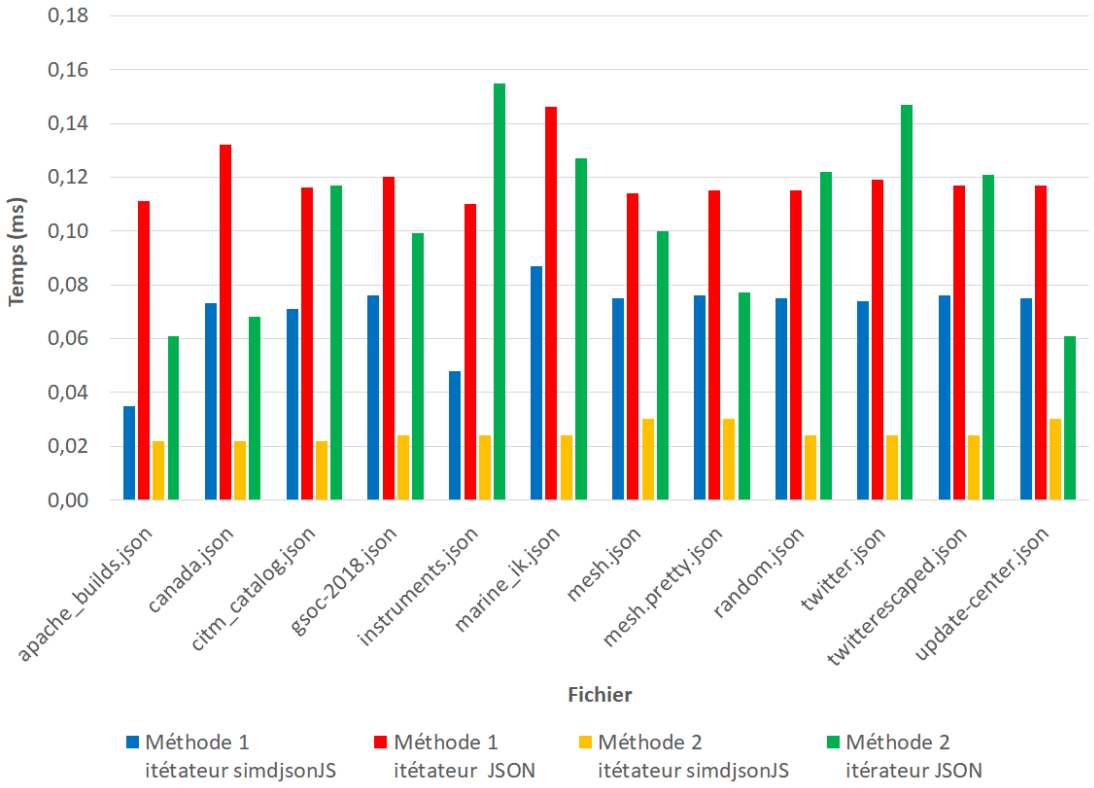

Figure 3.17: Performances, en millisecondes, du parcours d'objets

de type tableau nouvellement créé. Les figures 3.20 et 3.21 présentent le temps d'exécution des 100 premières itérations de la seconde méthode. Elles permettent de comparer le temps requis pour construire les éléments du tableau au temps requis lors des prochains appels visant à parcourir de nouveau le document.

L'analyse de ces résultats a permis de détecter une variation cyclique au niveau des performances du fichier numbers.json. Ce fichier, qui représente un tableau de 10 000 nombres décimaux, prend parfois plus de trois fois plus de temps pour la génération du parcours. La ligne de code qui crée cet effet a été isolée dans le code et correspond à la méthode utilisée dans les tests proposés par l'interface applicative N-API(Node.js, 2021c) permettant d'associer un tableau d'objets aux

protocoles d'itération JavaScript. Pour l'instant, aucune solution n'a été trouvée pour résoudre cette problématique.

Les autres résultats obtenus par la fonctionnalité implémentée pour la méthode 1 correspondent aux résultats attendus. En effet, le fichier github_events_1000.json représente le tableau le plus volumineux et est donc plus long à traiter, mais offre des temps plutôt stables. La génération de l'itérateur des plus petits fichiers offre aussi des performances stables et des temps plus acceptables.

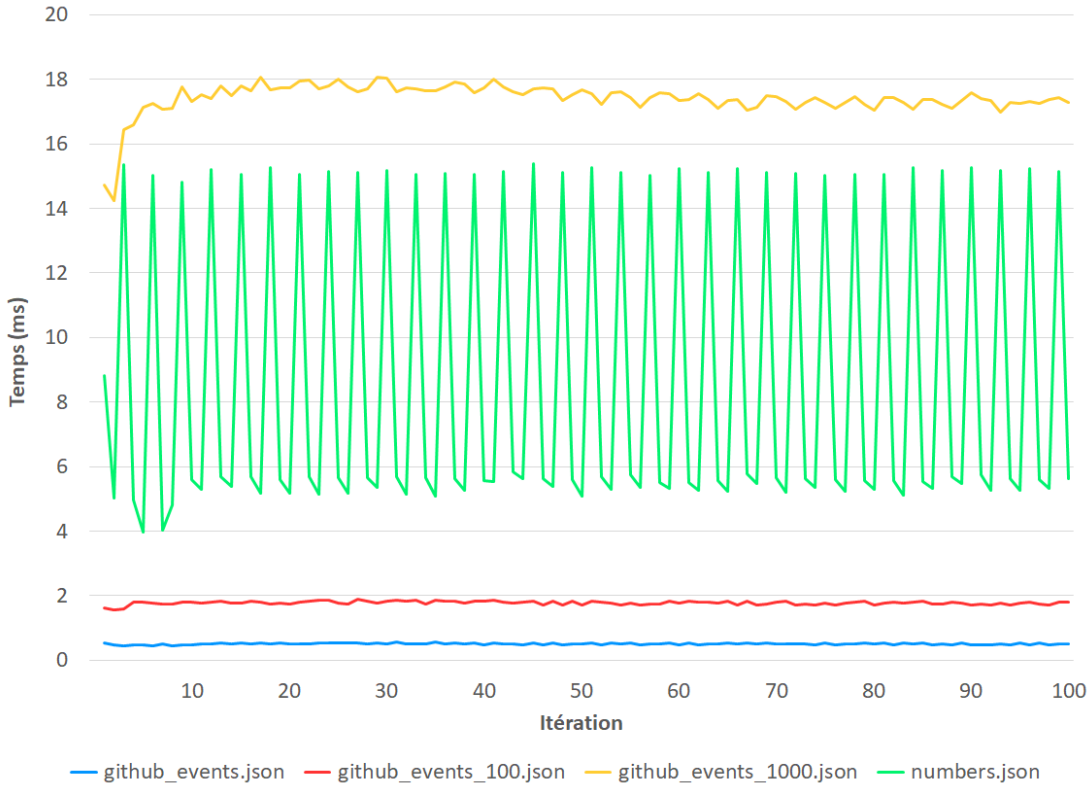

Figure 3.18: Performances, en millisecondes, des 100 premières itérations du parcours de la fonctionnalité implémentée et évaluée selon la méthode 1

L'analyse des performances de la fonction d'itération native est très intéressante. En effet, même si le document est recréé à chaque itération, seules les premières itérations sont plus lentes. Cela laisse supposer que des techniques de mise en

mémoire tampon sont être mises en place dans la fonctionnalité native afin conserver l'information et y avoir accès plus rapidement. Cela permet également de constater que le temps de génération maximal entre les 2 fonctionnalités est similaire, et toutes deux au-dessus de 10 millisecondes.

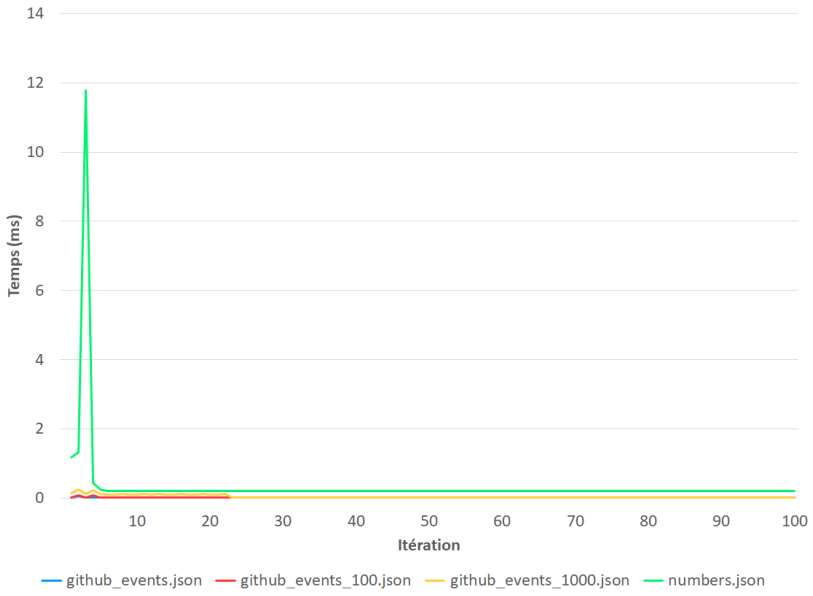

Figure 3.19: Performances, en millisecondes, des 100 premières itérations du parcours de la fonctionnalité native et évaluée selon la méthode 1

Les résultats obtenus par l'application de la seconde méthode démontrent des observations très similaires entre la fonctionnalité implémentée et son équivalent natif. La comparaison des performances permet d'identifier que la méthode implémentée et la méthode native tirent toutes deux avantage des appels subséquents.

Dans un contexte d'utilisation normale, l'utilisateur devra cependant d'abord analyser syntaxiquement le document avant d'en parcourir le contenu. Les deux méthodes d'analyse syntaxique les plus susceptibles d'être utilisées soit par passage du chemin d'accès en paramètre (méthode 2) et par passage du document sous forme de chaîne de caractères (méthode 3) sont comparées aux fonctionnalités

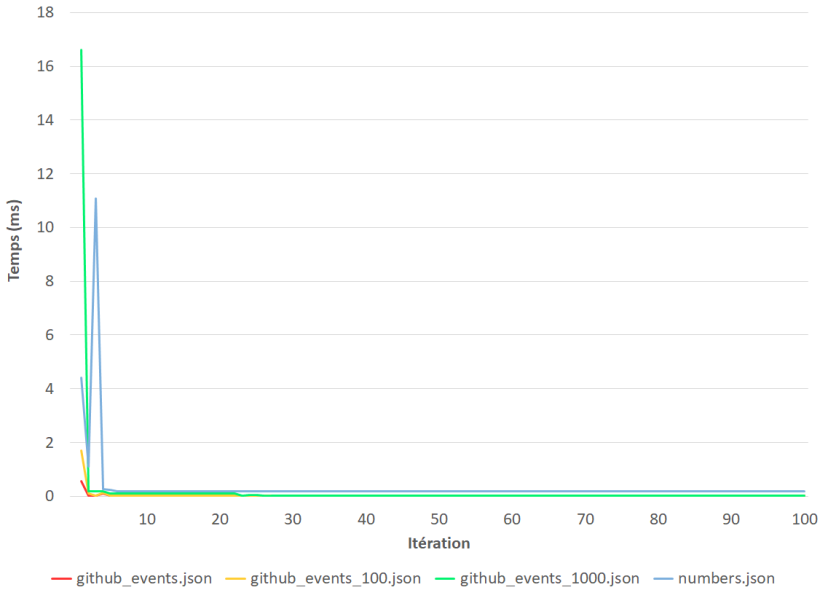

Figure 3.20: Performances, en millisecondes, des 100 premières itérations du parcours d'objet de la fonctionnalité implémentée et évaluée selon la méthode 2

natives équivalentes sur les fichiers représentant des objets structurés en tableaux. Les résultats présentés graphiquement à la figure 3.22

Les résultats observés permettent d'établir que pour ce type d'utilisation, il est préférable d'utiliser les fonctionnalités natives pour le moment. Il serait intéressant d'approfondir les recherches afin d'expliquer la variation des performances pour certains documents et s'il est possible de tendre vers les performances de la fonctionnalité native.

3.2.3.6 Recherche par requête

La spécification RFC 6901 (Bryan, 2013) définit une syntaxe standardisée permettant d'accéder à des objets spécifiques d'un document représentant un objet ou un tableau. Tel que défini dans la spécification, un pointeur JSON correspond à une chaîne de caractères contenant une série de jetons préfixés par le caractère /.

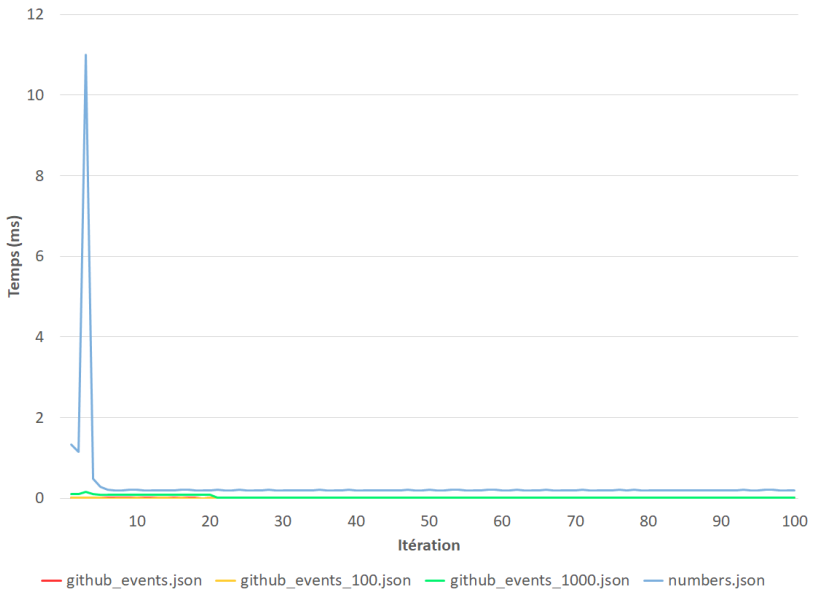

Figure 3.21: Performances, en millisecondes, des 100 premières itérations du parcours d'objet de la fonctionnalité native et évaluée selon la méthode 2

Chacun des jetons permet d'identifier la prochaine étape d'un parcours afin de récupérer l'élément recherché.

Les résultats des fonctionnalités mises en oeuvre ont été comparés au module json-pointer (Lemire, 2019a) offert par le gestionnaire de paquets npm. Le tableau 3.16 présente les résultats obtenus avec diverses requêtes sur différents documents structurés en objets ou en tableaux.

Afin de bien évaluer les performances, des requêtes permettant de récupérer des éléments au début et à la fin d'un objet ou d'un tableau ont été aussi été testées. Le tableau 3.17 présente les performances obtenues lors de diverses requêtes en utilisant le pointeur JSON, sous forme de tableau. La figure 3.23 présente les performances sous forme de graphique.

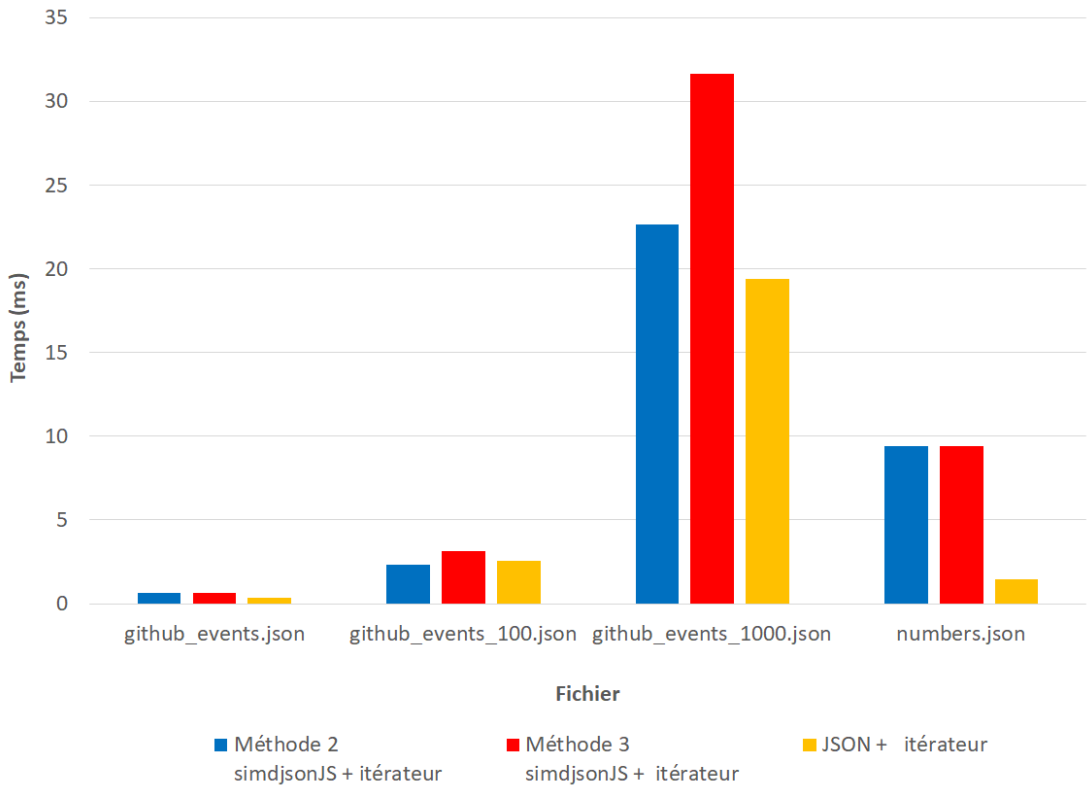

Figure 3.22: Performances, en millisecondes (ms), de l'appel des diverses méthodes d'analyse syntaxique combiné au parcours du document

Fichier	Requête	simdjson	npm/json-pointer
small/cars.json	0/tire_pressure/1	39.9	39.9
small/demo.json	Image/Width	800	800
small/demo.json	Image/Height	600	600
small/demo.json	Image/IDs/2	234	234
numbers.json	0	0.696468466152	0.696468466152
numbers.json	10000	0.763393189783	0.763393189783
numbers.json	10001	Attempted to access an element of a JSON array that is beyond its length.	Invalid reference token : 100001
nunbers.json	statuses/0/id	505874924095815700	505874924095815700
twitter.json	statuses/10/id	505874903094939650	505874903094939650
twitter.json	null	JSON Pointer parameter must be a string	Invalid reference token : null
twitter.json		The JSON field referenced does not exist in this object.	Invalid reference token :
twitter.json	qwerty	The JSON field referenced does not exist in this object.	Invalid reference token : qwerty

Tableau 3.16: Résultats obtenus lors de l'appel du pointeur JSON

Fichier	Requête	simdjson	npm/json-pointer
small/cars.json	0/tire_pressure/1	**0,003**	0,007
small/demo.json	Image/Width	**0,003**	*0,006*
small/demo.json	Image/Height	**0,003**	0,007
small/demo.json	Image/IDs/2	**0,003**	0,007
numbers.json	0	**0,003**	**0,005**
numbers.json	10000	0,015	**0,005**
numbers.json	10001	0,032	0,017
nunbers.json	statuses/0/id	*0,004*	0,008
twitter.json	statuses/10/id	**0,003**	**0,005**
twitter.json	null	0,015	0,02
twitter.json		0,021	0,019
twitter.json	qwerty	0,021	0,020

Tableau 3.17: Performances, en millisecondes, de l'appel du pointeur JSON

Les performances obtenues par la technique de recherche appuyée par la spécification RFC 6901 sont bonnes, les requêtes testées étant majoritairement plus rapides en utilisant la fonctionnalité implémentée. Les résultats présentés permettent cependant de constater que les performances diminuent lorsque le document en entier est parcouru et qu'il est alors plus performant d'utiliser le module équivalent offert pour traiter des objets JSON natifs.

3.2.3.7 Fonctionnalités natives non intégrées

Pour offrir une meilleure expérience pour l'utilisateur, d'autres fonctionnalités pourraient être intégrées au module proposé dans le cadre de cette expérimentation.

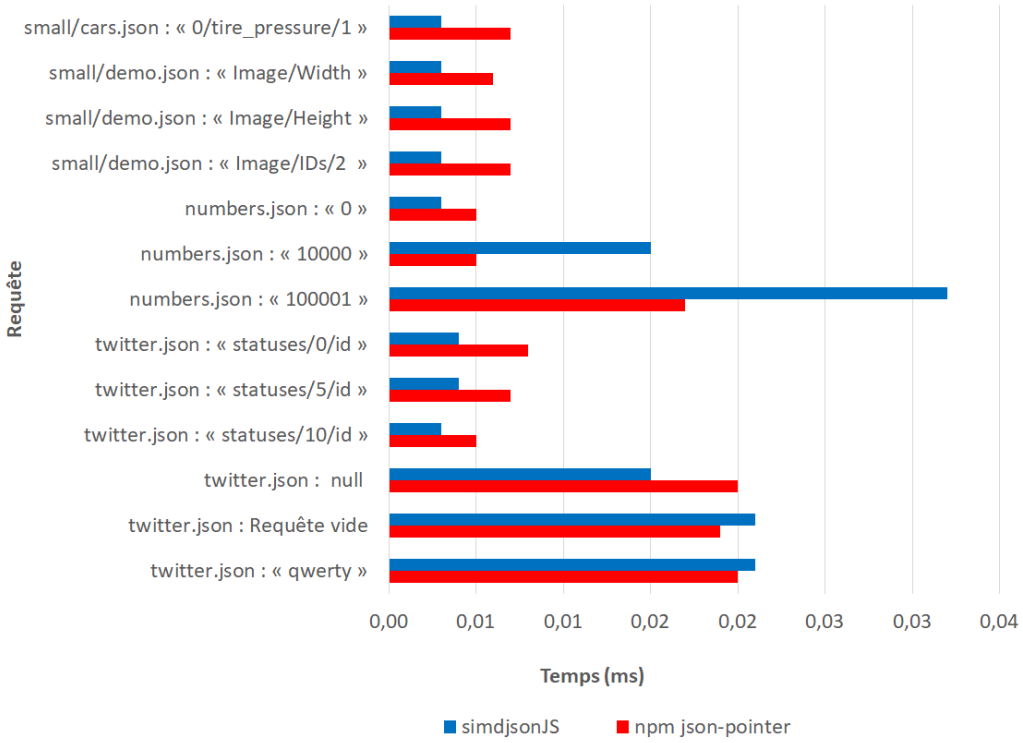

Figure 3.23: Performances, en millisecondes, de l'appel du pointer JSON

Par exemple :

— ajout d'une méthode permettant de valider si la structure d'un document est valide. Cette méthode pourrait également valider la structure en fonction d'un schéma fourni par l'utilisateur ;

— implémentation complète du prototype Array défini dans la documentation. JavaScript (MDN Web Docs, 2019a).

3.2.4 Restrictions et limitations

L'analyse des résultats permet également de se questionner sur le temps de conversion de types entre JavaScript et C++. Les performances moins intéressantes sont celles qui utilisent ces conversions pour obtenir les résultats désirés.

CONCLUSION

Les stratégies expérimentées visant à confirmer l'hypothèse sur la possibilité d'offrir des performances supérieures à celle de la fonction native `JSON.parse()` tout en conservant une utilisation standard ou équivalente de l'objet à l'aide du module N-API répondent aux objectifs fixés par cette recherche. Bien que les résultats ne soient pas tous excellents, ils demeurent bons en général. Les objets générés peuvent être parcourus de manière similaire à un objet JavaScript natif.

Les travaux réalisés dans le cadre de la première expérimentation ont permis de se familiariser avec la bibliothèque `simdjson` et l'interface applicative officielle de Node.js pour le développement de modules natifs N-API. Ils ont permis de relever des problèmes de gestion de mémoire de la version 1.6.2 du module node-addon-api. L'algorithme proposé a cependant des limitations. Lors du traitement de fichiers contenant des listes d'objets complexes, la vitesse d'exécution est inférieure à toutes les autres solutions d'analyse testées, ce qui a mené l'élaboration de la stratégie visée par la seconde expérimentation.

La seconde expérimentation est celle qui présente le gain de performances le plus intéressant au niveau de l'analyse syntaxique. Les performances sont de 3 à 7 fois plus rapides lorsque le chemin d'accès est passé en paramètres, et de 1.3 à 5 fois plus rapide lors du passage d'une chaîne de caractères, en comparaison à l'utilisation de la fonction d'analyse syntaxique native `JSON.parse()`. Elle minimise également les pertes de performances de l'appel des fonctions d'analyse syntaxiques proposées par la bibliothèque `simdjson`. Cette solution couvre les fonctionnalités principales (affichage, dénombrement, énumération et recherche). Combinée à

l'analyse syntaxique, la fonctionnalité permettant d'obtenir la liste des propriétés mise en oeuvre est plus performante que la combinaison des fonctionnalités natives, et ce, pour l'ensemble des fichiers évalués. La recherche à partir de critères définis par la spécification RFC 6901 JSON Pointer offre également des performances supérieures en utilisant la bibliothèque `simdjson` pour la majorité des requêtes évaluées.

La seconde expérimentation a également permis de soulever certaines limitations. Tout d'abord, la propriété permettant d'obtenir la taille offre des performances variables. Le parcours d'objet offre quant à lui des performances peu intéressantes lors d'un premier appel. Le peu de documentation sur les meilleures pratiques pour l'intégration de fonctions natives sur un objet encapsulé à l'aide de l'interface N-API a rendu plus difficile le développement des fonctionnalités essentielles. Pour en comprendre le principe, de nombreuses recherches ont été effectuées, et les stratégies employées ont été découvertes en consultant les tests présentés par l'interface (Node.js, 2021c). De plus, plusieurs fonctionnalités natives ne sont présentement pas offertes par la solution mise en oeuvre.

De futurs travaux pourraient rendre la solution proposée plus intéressante. Tout d'abord, il serait intéressant de valider s'il est possible d'optimiser le parcours d'un tableau dans le module N-API, et de valider si l'algorithme utilisé pour la recherche par requête de la bibliothèque `simdjson`, principalement au niveau du parcours d'indices dans les tableaux, est optimal. Il serait aussi intéressant d'intégrer les autres fonctionnalités du prototype Array. Par exemple, des fonctionnalités permettant de filtrer les résultats selon une requête offriraient de meilleures capacités dans l'analyse de données. Les méthodes `map()` et `reduce()` permettraient également de transformer les objets pour en extirper divers résultats essentiels lors de l'analyse de données massives. Il serait également intéressant de mesurer le gain de performances avec divers moteurs JavaScript et présenter les résultats obtenus

au niveau de l'analyse syntaxique, dans l'objectif d'améliorer les performances des appels à des API REST nécessitant l'utilisation de la méthode `JSON.parse()` offerte par ces moteurs.

Les technologies utilisées dans le cadre de cette recherche sont toutes d'actualité. Il est cependant probable que les technologies futures offriront de meilleures performances dans un avenir rapproché. Nous espérons que les résultats présentés dans ce mémoire contribueront à la recherche d'outils d'analyse syntaxique toujours plus performants.

ANNEXES

ANNEXE A

EXEMPLES DE MANIPULATION D'OBJETS

A.1 Exemple d'utilisation de l'extension simdjsonJS sous Node.JS

```
'use strict';

const { simdjson } = require('bindings')('addon');

const file_path = './jsonexamples/github_events.json';

const simdjsonOBJ = new simdjson({path : file_path});

// Affiche le contenu de l'objet
console.log(simdjsonOBJ);

// Affiche la taille de l'objet (tableau seulement)
console.log(simdjsonOBJ.length);

// Affiche les cles de l'objet
console.log(simdjsonOBJ.keys());

// Parcours de l'objet (tableau seulement)
try {
    for (let item of simdjsonOBJ) {
        console.log(item);
    }
} catch(error) {
    console.error(error);
}

// Parcours de l'objet et affichage de propriétés imbriquées spécifiques (tableau seulement)
try {
    for (let item of simdjsonOBJ) {
        console.log('actor : ');
        console.log(item.actor);
        console.log();

        console.log('repo : ');
        console.log(item.repo);
```

```
        console.log();

        console.log('payload.forkee.owner : ');
        if(item.payload && item.payload.forkee) {
            console.log(item.payload.forkee.owner);
        }
    }
} catch(error) {
    console.error(error);
}
```

A.2 Exemple d'utilisation d'un objet JSON natif sous Node.JS

```
'use strict';

const fs = require('fs');
const { simdjson } = require('bindings')('addon');

const file_path = './jsonexamples/github_events.json';

const file = fs.readFileSync(file_path, 'utf-8');
var JSONObj = JSON.parse(file);

// Affiche le contenu de l'objet
console.log(JSONObj);

// Affiche la taille de l'objet (tableau seulement)
console.log(JSONObj.length);

// Affiche les cles de l'objet
console.log(Object.keys(JSONObj));

// Parcours de l'objet (tableau seulement)
try {
    for (let item of JSONObj) {
        console.log(item);
    }
} catch(error) {
    console.error(error);
}

// Parcours de l'objet et affichage de propriétés imbriquées spécifiques (tableau seulement)
try {
    for (let item of JSONObj) {
        console.log('actor : ');
        console.log(item.actor);
        console.log();

        console.log('repo : ');
        console.log(item.repo);
        console.log();

        console.log('payload.forkee.owner : ');
        if(item.payload && item.payload.forkee) {
            console.log(item.payload.forkee.owner);
        }
    }
} catch(error) {
    console.error(error);
}
```

ANNEXE B

REPRÉSENTATION DES PROPRIÉTÉS

B.1 Fichier apache_builds.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
assignedLabels, mode, nodeDescription, nodeName, numExecutors, description,
↪  jobs, overallLoad, primaryView, quietingDown, slaveAgentPort,
↪  unlabeledLoad, useCrumbs, useSecurity, views
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
assignedLabels, mode, nodeDescription, nodeName, numExecutors, description,
↪  jobs, overallLoad, primaryView, quietingDown, slaveAgentPort,
↪  unlabeledLoad, useCrumbs, useSecurity, views
```

B.2 Fichier canada.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
type, features
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
type, features
```

B.3 Fichier citm_catalog.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
areaNames, audienceSubCategoryNames, blockNames, events, performances,
↪  seatCategoryNames, subTopicNames, subjectNames, topicNames,
↪  topicSubTopics, venueNames
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
areaNames, audienceSubCategoryNames, blockNames, events, performances,
↪  seatCategoryNames, subTopicNames, subjectNames, topicNames,
↪  topicSubTopics, venueNames
```

B.4 Fichier github_events.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
↪  22, 23, 24, 25, 26, 27, 28, 29
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
↪  22, 23, 24, 25, 26, 27, 28, 29
```

B.5 Fichier gsoc-2018.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
↪  22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
↪  40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
↪  58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
↪  76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
↪  94, 95, 96, 97, 98, 99, 100, 101, ..., 1100, 1101, 1102, 1103, 1104,
↪  1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1116,
↪  1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128,
↪  1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140,
↪  1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152,
↪  1153, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164,
↪  1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176,
↪  1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188,
↪  1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200,
↪  1201, 1202, 1203, 1204, 1205, 1206, 1207, 1208, 1209, 1210, 1211, 1212,
↪  1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224,
↪  1225, 1226, 1227, 1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236,
↪  1237, 1238, 1239, 1240, 1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248,
↪  1249, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260,
↪  1261, 1262, 1263
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
↪  22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
↪  40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
↪  58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
↪  76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
↪  94, 95, 96, 97, 98, 99, 100, ..., 1100, 1101, 1102, 1103, 1104, 1105,
↪  1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1116, 1117,
↪  1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129,
↪  1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141,
↪  1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1153,
↪  1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164, 1165,
↪  1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177,
↪  1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189,
↪  1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201,
↪  1202, 1203, 1204, 1205, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213,
↪  1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225,
↪  1226, 1227, 1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237,
↪  1238, 1239, 1240, 1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249,
↪  1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1261,
↪  1262, 1263
```

B.6 Fichier instruments.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
graphstate, instruments, message, name, orderlist, patterns, pluginstate,
↪  samples, version
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
graphstate, instruments, message, name, orderlist, patterns, pluginstate,
↪  samples, version
```

B.7 Fichier marine_ik.json

```
images, geometries, textures, metadata, materials, object, animations
```

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
images, geometries, textures, metadata, materials, object, animations}
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
images, geometries, textures, metadata, materials, object, animations
```

B.8 Fichier mesh.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
batches, morphTargets, positions, tex0, colors, influences, normals, indices
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
batches, colors, indices, influences, morphTargets, normals, positions, tex0
```

B.9 Fichier mesh.pretty.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
batches, colors, indices, influences, morphTargets, normals, positions, tex0
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
batches, morphTargets, positions, tex0, colors, influences, normals, indices
```

B.10 Fichier numbers.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
↪  22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
↪  40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
↪  58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
↪  76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
↪  94, 95, 96, 97, 98, 99, 100, 101, ..., 9950, 9951, 9952, 9953, 9954,
↪  9955, 9956, 9957, 9958, 9959, 9960, 9961, 9962, 9963, 9964, 9965, 9966,
↪  9967, 9968, 9969, 9970, 9971, 9972, 9973, 9974, 9975, 9976, 9977, 9978,
↪  9979, 9980, 9981, 9982, 9983, 9984, 9985, 9986, 9987, 9988, 9989, 9990,
↪  9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
↪  22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
↪  40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
↪  58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
↪  76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
↪  94, 95, 96, 97, 98, 99, 100, 101, ..., 9950, 9951, 9952, 9953, 9954,
↪  9955, 9956, 9957, 9958, 9959, 9960, 9961, 9962, 9963, 9964, 9965, 9966,
↪  9967, 9968, 9969, 9970, 9971, 9972, 9973, 9974, 9975, 9976, 9977, 9978,
↪  9979, 9980, 9981, 9982, 9983, 9984, 9985, 9986, 9987, 9988, 9989, 9990,
↪  9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000
```

B.11 Fichier random.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
id, jsonrpc, total, result
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
id, jsonrpc, total, result
```

B.12 Fichier twitter.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
statuses, search\_metadata
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
statuses, search\_metadata
```

B.13 Fichier twitterescaped.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
statuses, search\_metadata
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
statuses, search\_metadata
```

B.14 Fichier update-center.json

Propriétés obtenues en appelant la méthodes keys() sur un objet de type simdjson

```
connectionCheckUrl, core, id, plugins, signature, updateCenterVersion
```

Propriétés obtenues en appelant la méthodes Object.Keys() en passant en paramètre un objet JSON natif

```
connectionCheckUrl, core, id, plugins, signature, updateCenterVersion
```

RÉFÉRENCES

Angular (2020). angular/packages/common/http/src/xhr.ts. Récupéré le 12 janvier 2021 de https://github.com/angular/angular/blob/master/packages/common/http/src/xhr.ts

Angular (2021). Communicating with backend services using HTTP. Récupéré le 12 janvier 2021 de https://angular.io/guide/http

Axios (2021). axios. Récupéré le 12 janvier 2021 de https://github.com/axios/axios

Bryan, P. C. (2013). JavaScript Object Notation (JSON) Pointer. Récupéré le 12 janvier 2021 de https://tools.ietf.org/html/rfc6901

Docker (2021). Developers bring their ideas to life with Docker. Récupéré le 12 janvier 2021 de https://www.docker.com/why-docker

Facebook (2021a). react. *GitHub*. Récupéré le 12 janvier 2021 de https://github.com/facebook/react

Facebook (2021b). react / packages / react-fetch. Récupéré le 12 janvier 2021 de https://github.com/facebook/react/tree/master/packages/react-fetch

Fielding, R. T. et Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, *2*(2), 115–150.

Gallagher, N. (2017). How we built Twitter Lite. Récupéré le 12 janvier 2021 de https://blog.twitter.com/engineering/en_us/topics/open-source/2017/how-we-built-twitter-lite.html

jQuery (2021). jquery/src/ajax.js. Récupéré le 12 janvier 2021 de https://github.com/jquery/jquery/blob/master/src/ajax.js

Langdale, G. et Lemire, D. (2019). Parsing gigabytes of JSON per second. *CoRR*, *abs/1902.08318*. Récupéré le 12 janvier 2021 de http://arxiv.org/abs/1902.08318

Lemire, D. (2019a). json-pointer. Récupéré le 12 janvier 2021 de https://github.com/simdjson/simdjson/pull/217

Lemire, D. (2019b). More accurate number parsing. Récupéré le 12 janvier 2021 de https://github.com/simdjson/simdjson/pull/217

Lemire, D. (2019c). More accurate number parsing (217). Récupéré le 12 janvier 2021 de https://github.com/simdjson/simdjson/commit/e926b4b3c9b4af11bc126c49b4fefb2567cdbc9c

Lemire, D. (2019d). Parsing time is dominated by makeJSONObject (in the wrapper). Récupéré le 12 janvier 2021 de https://github.com/luizperes/simdjson_nodejs/issues/5

Li, Y., Katsipoulakis, N. R., Chandramouli, B., Goldstein, J. et Kossmann, D. (2017). Mison : a fast JSON parser for data analytics. *Proceedings of the VLDB Endowment*, *10*(10), 1118–1129.

Mclachlan, P. (2019). Introducing a Trusted Web Activity for Android. *Chromium Blog*. Récupéré le 12 janvier 2021 de https://blog.chromium.org/2019/02/introducing-trusted-web-activity-for.html

MDN Web Docs (2019a). Array. Récupéré le 12 janvier 2021 de https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array

MDN Web Docs (2019b). Array.length. Récupéré le 12 janvier 2021 de https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/length

MDN Web Docs (2019c). JSON.parse(). Récupéré le 12 janvier 2021 de https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/JSON/parse

MDN Web Docs (2019d). Les protocoles d'itération. Récupéré le 12 janvier 2021 de https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Les_protocoles_iteration

MDN Web Docs (2019e). Object.keys(). Récupéré le 12 janvier 2021 de https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Object/keys

MDN Web Docs (2021a). JavaScript. Récupéré le 12 janvier 2021 de https://developer.mozilla.org/fr/docs/Web/JavaScript

MDN Web Docs (2021b). Langage de programmation de haut niveau. Récupéré le 12 janvier 2021 de https://developer.mozilla.org/fr/docs/Glossaire/Langage_de_programmation_de_haut_niveau

MDN Web Docs (2021c). Manipuler des données JSON. Récupéré le 12 janvier 2021 de https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

Microsoft (2019). Microsoft / FishStore. Récupéré le 12 janvier 2021 de https://github.com/microsoft/FishStore

Node.js (2021a). Implications of ABI stability. Récupéré le 12 janvier 2021 de https://nodejs.org/api/n-api.html#n_api_implications_of_abi_stability

Node.js (2021b). node-addon-api Changelog. Récupéré le 12 janvier 2021 de https://github.com/nodejs/node-addon-api/blob/master/CHANGELOG.md

Node.js (2021c). node-addon-api/test/objectwrap.cc. Récupéré le 12 janvier 2021 de https://github.com/nodejs/node-addon-api/blob/e8935bd8d97ba61cd4c005fab571d9fb39ab9249/test/objectwrap.cc

Node.js (2021d). Node.js v14.7.0 Documentation. Récupéré le 12 janvier 2021 de https://nodejs.org/api/n-api.html

npm trends (2021). angular vs react vs vue. Récupéré le 12 janvier 2021 de https://www.npmtrends.com/angular-vs-react-vs-vue

Patterson, D. A. et Hennessy, J. L. (2016). *Computer Organization and Design ARM Edition : The Hardware Software Interface*. Morgan kaufmann.

Peres, L. F. (2019). simdjson_nodejs. Récupéré le 12 janvier 2021 de https://github.com/luizperes/simdjson_nodejs/blob/master/README.md

Peres, L. F. (2021). simdjson_nodejs. Récupéré le 12 janvier 2021 de https://github.com/luizperes/simdjson_nodejs

Pflug, K., Chinnathambi, K., Gustafson, A. et Shahid, I. (2018). Welcoming Progressive Web Apps to Microsoft Edge and Windows 10. Récupéré le 12 janvier 2021 de https://blogs.windows.com/msedgedev/2018/02/06/welcoming-progressive-web-apps-edge-windows-10/

Rauschmayer, A. (2014). How javascript was created in speaking javascript.

Raymond, E. S. (2003). *The art of Unix programming*. Addison-Wesley Professional.

React (2021). AJAX et les API. Récupéré le 12 janvier 2021 de https://fr.reactjs.org/docs/faq-ajax.html

Rigtorp, E. (2021). Awesome Modern C++. Récupéré le 12 janvier 2021 de https://awesomecpp.com

Schulhof, G., Dawson, M., Schlight, J. et Gobbo, N. D. (2019). Next Generation N-API : A hands-on workshop. Récupéré le 12 janvier 2021 de https://static.sched.com/hosted_files/njsi2019/ee/N-API-Workshop.pdf

simdjson (2021a). simdjson / simdjson. Récupéré le 12 janvier 2021 de https://github.com/simdjson/simdjson

simdjson (2021b). The simdjson library. Récupéré le 12 janvier 2021 de https://simdjson.org/

Stack Overflow (2020). 2020 Developer Survey. Récupéré le 12 janvier 2021 de https://insights.stackoverflow.com/survey/2020

Vue.js (2019). Utiliser Axios pour consommer des API. Récupéré le 12 janvier 2021 de https://fr.vuejs.org/v2/cookbook/using-axios-to-consume-apis.html

W3Techs (2021). Usage statistics of Node.js. Récupéré le 12 janvier 2021 de https://w3techs.com/technologies/details/ws-nodejs

Warren, T. (2019). Microsoft has turned Outlook into a Progressive Web App. Récupéré le 12 janvier 2021 de https://www.theverge.com/2019/11/26/20983886/microsoft-outlook-com-pwa-progressive-web-app-install-features

Wikipédia (2021). Single instruction multiple data. Récupéré le 12 janvier 2021 de https://fr.wikipedia.org/wiki/Single_instruction_multiple_data